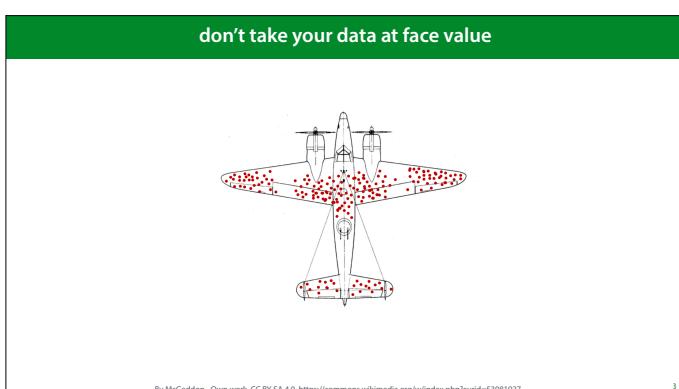


|section|Missing values and outliers|  
|video|<https://surfdrive.surf.nl/files/index.php/s/v4BiYSB6CIEjvpk/download>|



To motivate this lecture, let's look at a famous historical case of operations research. In the second World War, the allies executed many bombing runs, and often, their planes came back looking like this.

To investigate where they should reinforce their planes, investigators made a tally of the most common points on the plane they were seeing damage.



By McGeddon - Own work, CC BY-SA 4.0, <https://commons.wikimedia.org/w/index.php?curid=53081927>

3

The initial instinct was to reinforce those places that registered the most hits.

However, it was soon pointed out (by a man called Abraham Wald) that this ignores a crucial aspect of the *source* of the data. They weren't tallying where planes were most likely to be hit, they were tallying where planes were most likely to be hit *and come back*.

The places where they weren't seeing *any* hits were exactly the places that should be reinforced, since the planes that were hit there didn't make it back.

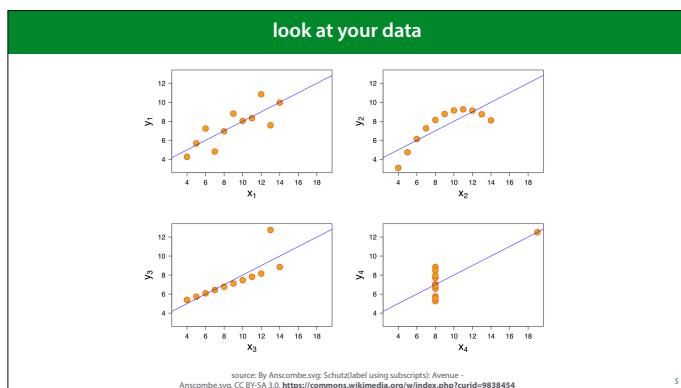
This specific effect is called **survivorship bias**, and it's worth keeping in mind, but the more general lesson for today, is that you should **not take your data at face value**. Don't just load your data into an ML model and check the predictive performance: consider what you're ultimately trying to achieve, and consider how the source of your data will affect that goal.

By McGeddon - Own work, CC BY-SA 4.0, <https://creativecommons.org/licenses/by-sa/4.0/>

look at your data									
	mean of x	mean of y	var. of x	var. of y	correlation between x and y	regression line w	regression line b	r <sup>2</sup>	
dataset 1	9	7.5	11	4.125	0.816	0.5	3	0.67	
dataset 2	9	7.5	11	4.125	0.816	0.5	3	0.67	
dataset 3	9	7.5	11	4.125	0.816	0.5	3	0.67	
dataset 4	9	7.5	11	4.125	0.816	0.5	3	0.67	

Imagine that I gave you four datasets, each with two features x and y. For all datasets all of the following statistics give the same value: the mean and variance of x, the mean and variance of y, the correlation between x and y, the parameters of the linear regression line that best fits, and the r<sup>2</sup> of the correlation.

You would conclude that the datasets must be pretty similar, right?

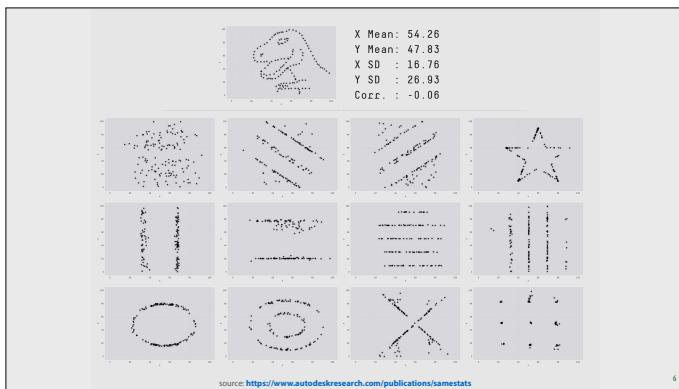


One important aspect of not taking the data at face value is to *look at it*.

These are the four datasets from the previous slide. They are a common example, called Anscombe's quartet. Only when we look at the datasets, do we see how different they are.

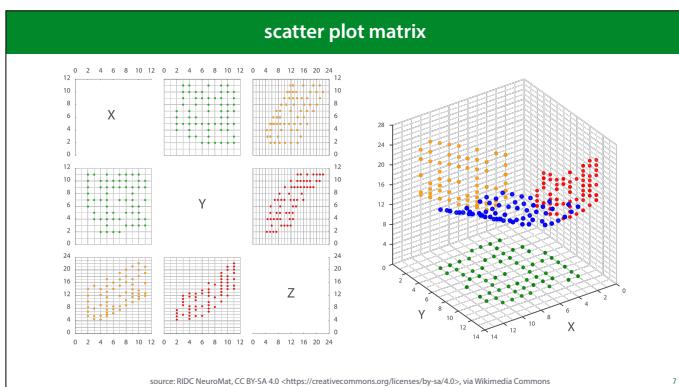
More importantly, only when we look at the data, do we see the patterns that define them. These are the patterns we want to get at if we want to understand the data. And none of them are revealed by the descriptive statistics of the previous slide.

source: By Schutz: Avenue - Anscombe.svg, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=9838454>



Here is a more modern variant: the datasaurus dozen.

Recommended reading: <https://www.autodeskresearch.com/publications/samestats>



In machine learning and data science, our datasets are rarely two-dimensional, so we don't have the luxury of simply doing a scatter plot. Looking at our data, in a way that provides insight almost always requires a lot of ingenuity and creativity.

For high-dimensional, multivariate data, of the kind we've been dealing with so far, a good place to start is to produce a **scatter plot matrix**. This is simply a large grid of every scatter plot you can produce between any two features in your data. Often, only the plots below or above the diagonal are shown. The scatterplot matrix gives you a good idea of how the features relate to each other.

If you have a target value (a class or a regression target), it's a good idea to include it among the features for the scatter plot matrix. That way, you can see what relation each feature has with the target in isolation from the other features.

On the right, we see the data as a 3D point cloud (in blue), together with the three projections to 2d (in yellow red and green) that the scatterplot matrix gives us.

source: RIDC NeuroMat, CC BY-SA 4.0 <https://creativecommons.org/licenses/by-sa/4.0/>, via Wikimedia Commons

## cleaning your data

- Missing data:  
missing labels  
missing values
- Outliers
- Class imbalance

In the rest of this video, we'll look at ways you can clean up your data, to make it useable for a classification or a regression task.

### missing values

income	status	unemployed
32000	married	true
	single	false
89000		true
34000	divorced	false
54000	married	true
		false
21000		true
25000	single	true

We'll start with missing data. Quite often, your data will look like this.

You will need to do something about those gaps, before any machine learning algorithm will accept this data.

### missing labels

income	status	unemployed
32000	married	true
2000	single	
89000	single	true
34000	divorced	false
54000	married	
34000	married	false
21000	divorced	
25000	single	true

What approach you should take is different, depending on whether values from the feature columns are missing, or values from the target column are missing.

## simple solutions

- Remove the feature
- Remove the instances
  - are the data missing **uniformly**?

If you have missing values in one of your features, the simplest way to get rid of them is to just remove the feature(s) for which there are values missing. If you're lucky, the feature is not important anyway.

You can also remove the instances with missing data. Here you have to be careful. If the data was not corrupted uniformly, removing rows with missing values will change your data distribution.

For example, you might have data gathered by volunteers in the street using some electronic equipment. If the volunteer in Amsterdam had problems with their hardware, then their data will contain missing values, and the collected data will not be representative of Amsterdam.

Another way you might get non-uniformly distributed missing data is if your data comes from a questionnaire, where people sometimes refuse to answer certain questions. For instance, if only rich people refuse to answer questions about their taxes, removing these instances will remove a lot of rich people from your data and give you a different distribution.

**How can you tell if data is missing uniformly?** There's no surefire way, but usually you can get a good idea by plotting a histogram of how much data is missing against some other feature. For instance if the number of instances with missing features against income is very different from the regular histogram over income, you can assume that your data was not corrupted uniformly.

Of course it also helps if you can work out why your data has missing values. Again, don't take the data at face value, look into where it came from, and what the details of that process can tell you.

Think about the **REAL-WORLD** use case.

Let's zoom out a little before we move on. Whenever you have questions about how to approach something like this, it's best to think about the **real-world setting** where you might apply your trained model. We often call this "production", a term used in software development for the system that will be running the final deployed version of the software. Some machine learning models literally end up in a production environment, but we might also use machine learning models to perform business analytics, clinical decision support or in a scientific experiment. Wherever your model is meant to end up after you've finished your experimentation, that's what we'll call **production**.

And production is what you're trying to simulate, when you train your model and test it on a test set. So the choices you make, should make your experiment as close of a *simulation* of your production setting as you can manage.

For example, in the case of missing values, the big question is: **can you expect missing data in production?** Or, will

your production data be clean, and are the test data just noisy because the production environment isn't ready yet?

Examples of production systems that should expect missing data are situations where data comes from a web form with optional values or situations where data is merged from different sources (like online forms and phone surveys).

You may even find yourself in a situation where the test data has no missing values (since it was carefully gathered) but the production system *will* have missing values (because there, the data will come from a web form). In that case, you may want to introduce missing values artificially in your test data, to simulate the production setting and study the effect of missing data.

So remember, whenever you're stuck on how to process your data: think what the production setting is that you're trying to simulate, and make your choices based on that.

### will you get missing values in production?

YES:

Keep them in the [test set](#), and make a model that can deal with them them.

NO:

Endeavour to get a [test set without missing values](#), and test different methods for completing the data in the [training set](#) only.

If you expect to see missing in production, then your model needs to be able to *consume* missing values, and you should keep them in the test set. For categorical features, the easiest way to do this is to add a category for missing values. For numerical features, we'll see some options in the next slide.

If your production setting won't have missing values, then that's the setting you want to simulate. If at all possible, you should get a test set without missing values, even if the training set has them. You can then freely test what method of dealing with the missing training values gives the best performance on the test set.

*Or rather, on the validation set first, and then later on the test set. For now, we'll talk about test set performance to keep things simple, and take it as read, that you will use a validation set as a proxy to guard against multiple testing.*

If you cannot get a test set without missing values, one thing you can do is to report performance on both the data that has the instances with missing values removed and the data that has the missing values filled in by some mechanism. Neither are ideal simulations of the production setting, but the combination of both numbers hopefully gives you some idea.

## imputation: guess the missing values

categorical data: use the mode

numerical data: use the mean

make the feature a target value and train a model  
kNN, linear regression, etc.

At some point, either in the **training data** or the **test data**, we will probably need to fill in the missing values. This is called **imputation**.

A simple way to do this in categorical data is to use the **mode**, the most common category. For numerical data, the **mean** or **median** are simple options. We'll look at when you should use which later in this video.

A more involved, but more powerful way, is to **predict the missing value** from the other features. You just turn the feature column in to a target column and train a classifier for categoric features, and a regression model for numeric features.

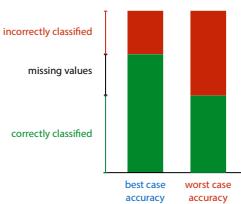
## missing labels

### training set

- train only on labeled data
- impute the missing labels
- many missing labels: semi-supervised learning

### test set

- don't impute, don't ignore!
- report your uncertainty

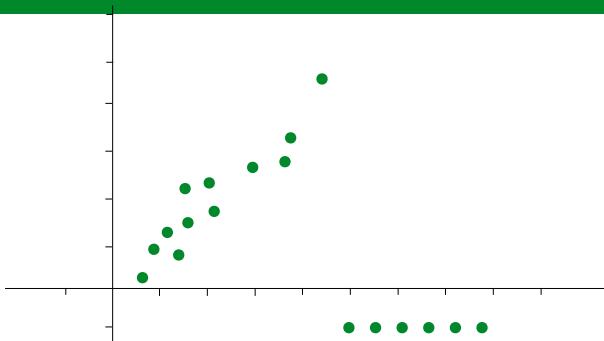


If your target label has missing values, the story is a little different. In the **training set** you are free to do whatever you think is best. You can remove instances, or impute the missing labels. If you have a lot of missing labels, this essentially becomes a semi-supervised learning setting as we saw in the first lecture.

On the **test set** however, you shouldn't impute or ignore the missing values, since that changes the task, and most likely makes it easier, which will give you false confidence in the performance of your model. Instead, you should **report the uncertainty** created by the missing values.

In classification, this is easy: you compute the accuracy under the assumption that your classifier gets all missing values correct and under the assumption that it gets all missing values wrong: this gives you a **best case** and a **worst case** scenario, respectively. Your true accuracy on the test set is somewhere in between.

## unnatural outliers



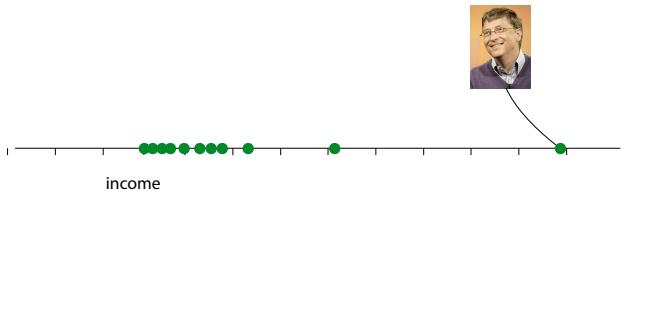
Another problem that we need to worry about, is **outliers**. Values in our data that take on unusual and unexpected values.

Outliers come in different shapes and sizes. The most important question is whether your outliers are natural or unnatural.

Here, the six dots to the right are so oddly, mechanically aligned that we are probably looking at some measurement error. Perhaps someone is using the value -1 for missing data.

We can remove these, or interpret them as missing data, and use the approaches just discussed.

## natural outliers



In other cases, however, the “outlier” is very much part of the distribution. This is what we call a **natural outlier**. Bill Gates may have a million times the net worth of anybody you are likely to meet in the street, but that doesn’t mean he isn’t part of the distribution of income.

If we fit a normal distribution to this data, the outlier would ruin our fit, but that’s because the data *isn’t normally distributed*. What we should do is recognize that fact, and adapt our model, for instance by removing assumptions of normally distributed data.



Here’s a metaphor for natural and unnatural outliers. If our instances are image of faces, the image on the left, that of comedian Marty Feldman, is an extreme of our data distribution. It looks unusual, but it’s crucial in fitting a model to this dataset. The image on the right is clearly corrupted data. It tells us nothing about what human faces might look like, and we’re better off removing it from the data.

However, remember the real-world use-case: if we can expect corrupted data in production as well, then we should either train the model to deal with it, or make the clean-up automatic, so we can perform it in production as well. This would require us to have some way to detect automatically, whether something is an outlier. If the outliers are rare, and we have a lot of data, it may be easier just to leave them in and hope the model can learn to work around them, even if they are unnatural outliers.

## outliers

Are they mistakes?

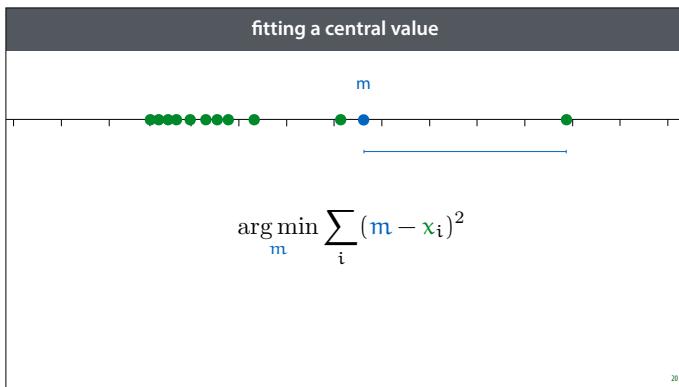
- Yes: deal with them.
- No: leave them be. Check your model for strong assumptions of normality.

Can we expect them in production?

- Yes: Make sure the model can deal with them.
- No: Remove. Get a [test set](#) that represents the production situation.

If you have very extreme values that are not mistakes (like Bill Gates earlier), your data is probably not normally distributed. If you use a model which assumes normally distributed data, it will be very sensitive to these kinds of “outliers”. It may be a good idea to remove this assumption from your model (or replace it by an assumption of a heavy-tailed distribution).

Note that you have to know your model really well to know if there are assumptions of normality. For instance anything that uses *squared errors* essentially has an assumption of normality.



To illustrate: let's learn which *single value* best represents our data. We choose a value  $m$ , compute the distance to all our data points (the residuals) and try to minimise their squares.

We can use such a single value for imputation to replace missing values or outliers, but this is also a kind of simplified picture of linear regression: if we had a regression problem with no features, the best we could do is predict the same single value for all instances. Which value should we pick to minimize the squared errors?

$$\arg \min_m \sum_i (m - x_i)^2$$

$$\sum_i \frac{\partial(m - x_i)^2}{\partial m} = 0$$

$$\sum_i \frac{\partial(m - x_i)^2}{\partial m - x_i} \frac{\partial m - x_i}{\partial m} = 0$$

$$2 \sum_i m - x_i = 0$$

$$n m - \sum_i x_i = 0$$

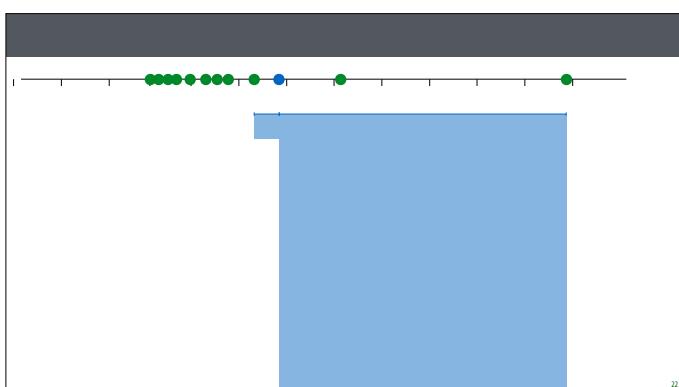
$$m = \frac{\sum_i x_i}{n}$$

21

We take the derivative of the objective function and set it equal to zero. No gradient descent required here, we'll just solve the problem analytically.

What we find is that the optimum is **the mean**. The assumption of squared errors leads directly to the use of the mean as a *representative example* of a set of points.

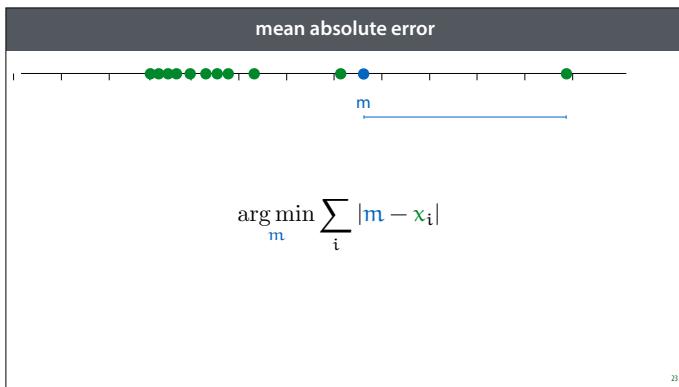
*Remember the previous lecture where we showed that the maximum likelihood objective on the normal distribution led to a least squares objective for its parameter  $\mu$ ? You can tie that to this derivatiation, and conclude that the maximum likelihood solution for  $\mu$  is the arithmetic mean of the data.*



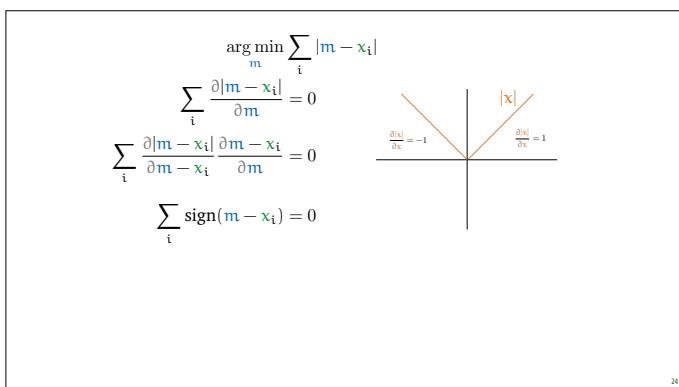
We can now see why the the assumption of squared errors is so disastrous in the case of the income distribution.

If Bill Gates makes a million times as much as the next person in the dataset, he is not pulling on the mean a million times as much, he's pulling  $10^{12}$  times as much.

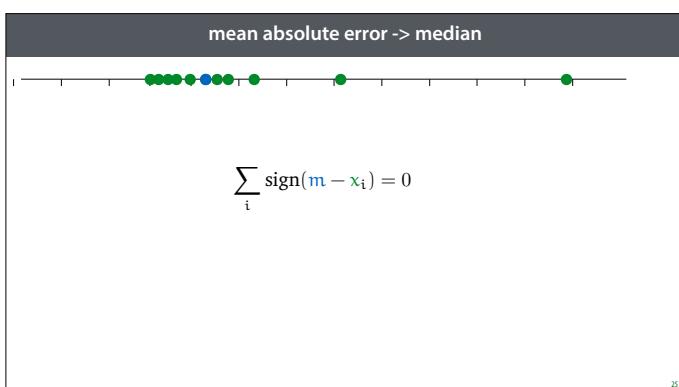
Hence the joke: A billionaire walks into a homeless shelter and says "What a bunch of freeloaders, the average wealth in this place is more than a million dollars!"



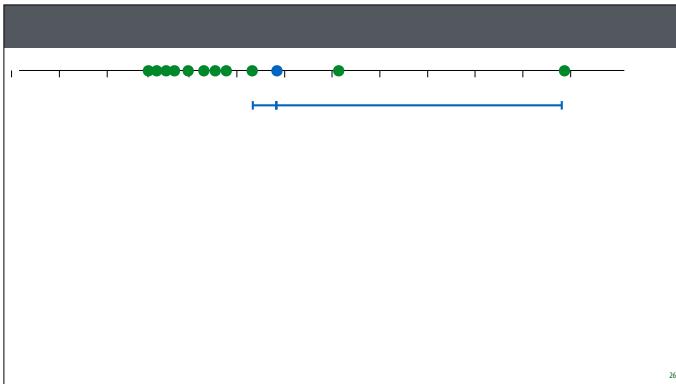
To get rid of the normality assumption, or rather, replace it by another assumption, we can use the **mean absolute error** instead. We take the residuals, but we sum their *absolute value* instead of their *squared value*. Which is the most representative value that minimises *that* error?



To work this out, we need to know the derivative of the absolute function. This function is the identity if the argument is positive (so its derivative is 1) and the negative identity if the argument is negative



We've worked out that the value  $m$  that minimizes this error is the one for which the signs of the residuals sum to zero. This happens if the sum contains as many "-1"s as "+1"s, that is, if we have as many instances to the left of  $m$  as we have to the right. In other words, **the median minimizes the mean absolute error**.



If we use the median, Bill Gates still has a strong pull on the our representative value  $m$ , but it's proportional to his distance to  $m$ , not to the square of the distance.

[Article](#) [Talk](#)

[Read](#) [Ed](#)

**List of countries by wealth per adult**

From Wikipedia, the free encyclopedia

By country

Median and mean wealth per adult, in US dollars. Countries and subnational areas.

Initially in rank order by median wealth. (2019 publication)<sup>[1]</sup>

Rank	Country or subnational area	Median wealth per adult, US dollars	Mean wealth per adult, US dollars	Adult population, thousands
1	Switzerland	227,891	564,653	6,866
2	Hong Kong	146,867	489,258	6,267
3	United States	65,904	432,365	245,140
4	Australia	181,361	386,058	18,655
5	Ireland	165,861	380,866	250
6	Luxembourg	139,769	358,003	461
7	New Zealand	116,433	304,124	3,525
8	Singapore	98,967	297,873	4,837
9	Canada	107,004	294,255	29,136
10	Denmark	88,784	284,022	4,475
11	United Kingdom	97,455	280,049	51,209
12	Netherlands	91,057	279,077	13,326
13	France	101,942	276,121	49,722
14	Austria	94,070	274,919	7,092
15	Ireland	104,842	272,310	3,491

27

This mistake, of using the mean when a normal distribution is not an appropriate assumption, is sadly very common.

For example, you might hear someone say something like “there’s no poverty in the US, it’s the third richest country in the world by average personal wealth”.

Wikipedia allows us to fact-check this quickly and it is indeed true. But remember that Bill Gates and Jeff Bezos live in the US, and as we saw, such people have a pretty strong pull on the mean. Luckily, Wikipedia also allows us to sort the same list by *median* wealth.

[Article](#) [Talk](#)

[Read](#) [Ed](#)

**List of countries by wealth per adult**

From Wikipedia, the free encyclopedia

By country

Median and mean wealth per adult, in US dollars. Countries and subnational areas.

Initially in rank order by median wealth. (2019 publication)<sup>[1]</sup>

Rank	Country or subnational area	Median wealth per adult, US dollars	Mean wealth per adult, US dollars	Adult population, thousands
1	Switzerland	227,891	564,653	6,866
2	Australia	181,361	386,058	18,655
3	Ireland	165,861	380,866	250
4	Hong Kong	146,887	489,258	6,267
5	Luxembourg	139,769	358,003	461
6	Belgium	117,003	246,135	8,913
7	New Zealand	116,433	304,124	3,525
8	Japan	110,408	298,104	104,963
9	Canada	107,004	294,255	29,136
10	Ireland	104,842	272,310	3,491
11	France	101,942	276,121	49,722
12	United Kingdom	97,455	280,049	51,209
13	Australia	94,070	274,919	7,092
22	United States	65,904	432,365	245,140
15	Austria	94,070	274,919	7,092

28

If we do that, we see that the US suddenly drops to 22nd place. This drop indicates how big the income inequality is.

*The Netherlands drops from 12th to 34, incidentally. So there's plenty of income inequality over here as well.*

 GamesIndustry @GIBiz

"There are 220,000 or so people employed in the US video game business. They make about \$100,000 on average, maybe more. It's hard to imagine what would motivate that crew to unionize"

Strauss Zelnick talks unions, release planning, next-gen, E3 and more



29

Here's an example of this fallacy in the wild. In 2019, there was a discussion in the US about unionisation in the games industry. Here, one of the heads of Take-Two suggests that because the *average* yearly salary has six figures, unions are unlikely.

Whether rich people can benefit from unions is a question for a different series of lectures, but the fact that the average wages are high, most likely just means that there is a small number of very rich people in the industry. We'd need to know the *median* to be sure.

source: <https://twitter.com/GIBiz/status/1140900959322804224?s=20>

### models that can deal with natural outliers

Beware of squared errors (MSE)  
They are only justified if your values have well-determined scale

*lecture 8*

Model noise with a heavy-tailed distribution  
Choose the median over the mean, MAE over MSE.

The proof is in the pudding.  
The performance on the test/validation set will be the deciding factor.

30

If you want to adapt your model to deal with natural outliers, beware of hidden assumptions of normality. We saw a hint last lecture already that assuming normality leads to squared error objectives. We'll look at this in detail in a later lecture. For now, keep this link in mind when you look at the distribution of your outliers.

Consider modelling your noise with a heavy-tailed distribution instead, in other words, one which makes outliers more likely. Using the median instead of the mean is one way to do this.

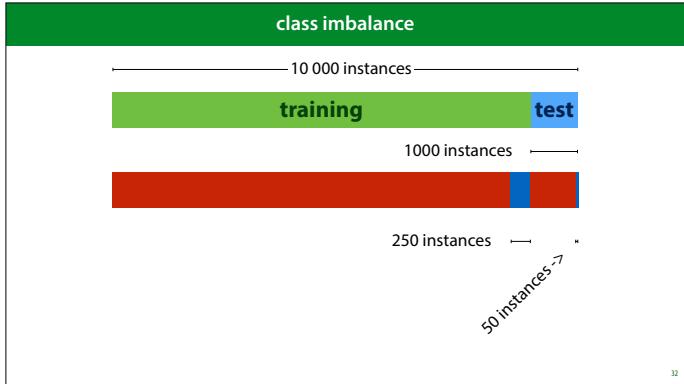
If you are doing regression and your target label is non-normally distributed then you can use the sum of absolute errors as a loss function instead of the sum of squared errors. This will also implicitly assume a more heavy-tailed distribution than the normal, but even more heavy tailed distributions are available. We'll look a little bit more at modeling data with different distributions in later lectures.

### Data pre-processing

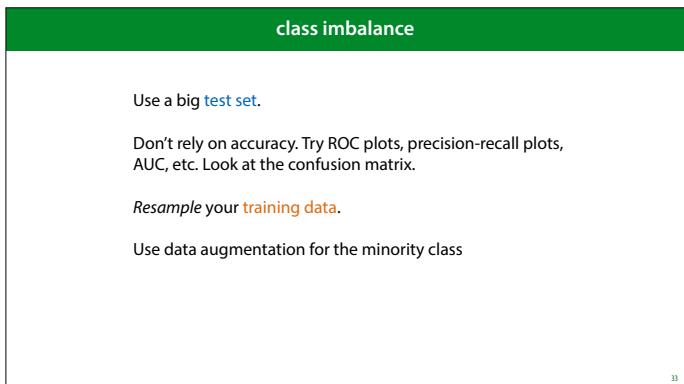
Part 2: Class imbalance and feature design

Machine Learning  
mlvu.github.io  
Vrije Universiteit Amsterdam

|section|Class imbalance and feature design  
|video|<https://surfdrive.surf.nl/files/index.php/s/o5yQd1mIGCsF8SP/download>



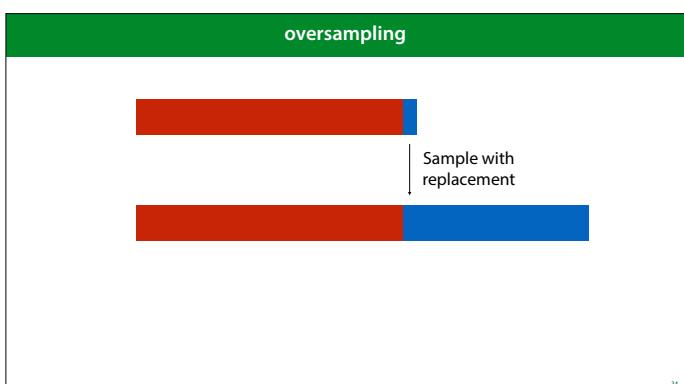
In the last lecture, we saw that class imbalance can be a big problem. We know what we can do to help our analysis of imbalanced problems, but how do we actually improve training?



We will assume that the class imbalance will happen in production as well as in your experiments. That means your **test set** needs to have the class imbalance in it, to be a fair simulation of the production setting. You can fiddle around with the training data all you like, but the **test data** (and by extension the **validation data**) needs to represent the natural class distribution.

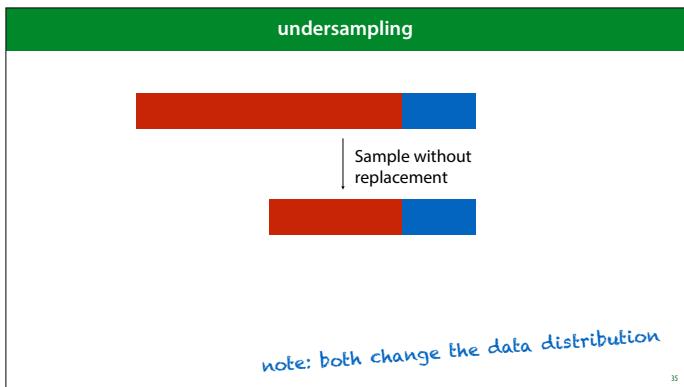
As a result, you'll need to focus on getting a large test set even more than normal: your problem is essentially that of detecting instances of the minority class. If you only have 25 of them in your test set, you won't get a very good idea of how well your classifier can detect them, even if you have 10000 majority class instances.

This is usually a painful step, since withholding a lot of test data leaves you with very little training data. However, since you are allowed to manipulate the training data however you like, you should prioritize the test data. There may be clever tricks to get more mileage out of the imbalanced training data, but without a proper test data, you won't be



The most common approach is to **oversample** your minority class by sampling with replacements. That is, you create a new dataset that is bigger than the original one, by adding a number of copies of randomly sampled instances of the minority class. That way, the class balance is more even in your new dataset.

The advantage is that this leads to more data. The disadvantage is that you end up with duplicates in your dataset. This may increase the likelihood of overfitting, depending on what algorithm you are using.



You can also **undersample** your majority class. You create a copy of your dataset where some randomly selected instance of the majority class are thrown out.

This doesn't lead to duplicates, but it does mean you're throwing away data.

If you have an algorithm that makes multiple passes over the dataset (like gradient descent) it can help to resample the dataset again for every pass, so that you don't lose the variation in the original data.

Whether you oversample or undersample, you should be aware that you are changing the class distribution in the data. If you increase the proportion of the majority class, the classifier will be more likely to classify things by the minority class. This will be a tradeoff: you want to oversample to the point where the classifier begins to pick up on the features that indicate the minority class, but not so much that the classifier begins seeing the minority class when the features do not indicate it.

The simplest way to achieve this is to treat the amount of resampling as a hyperparameter, and to optimize for precision/recall or ROC curves.

*Another option is to treat the classifier as a ranking classifier, as we saw in an earlier lecture. That way, we can train on re-balanced data, and then move the threshold after training to tune how eager the classifier is to call something positive.*

data augmentation
<p>SMOTE: add midpoints between nearby minority class data points.</p> <p>Domain specific: rotate or translate images in the minority class. Add Gaussian noise.</p> <p>Remember: only on the <b>training data</b>. Keep the <b>test data</b> as is.</p> <p>more information: <a href="https://www.kaggle.com/rafjaa/resampling-strategies-for-imbalanced-datasets">https://www.kaggle.com/rafjaa/resampling-strategies-for-imbalanced-datasets</a></p>

A more sophisticated approach is to oversample the minority class with **new data** derived from the existing data.

SMOTE is a good example: it finds small clusters of points in the minority class, and generates their mean as a new minority class point. This way, the new point is not a duplicate of any existing point, but it is still in a region that contains a lot of points in the minority class, to keep it realistic.

We don't have time to go into this deeply. If you run into this problem in your project, click the link for detailed explanation.

more information: <https://www.kaggle.com/rafjaa/resampling-strategies-for-imbalanced-datasets>

## getting features

phone nr	income	status	unemployed	birthdate	age
0646785910	32000	married	true	4-5-78	41
0207859461	45000	single	false	3-6-00	19
0218945958	89000	married	true	4-7-91	28
0645789384	34000	divorced	false	3-11-94	25
0652438904	54000	married	true	21-3-95	24
0309897969	36000	single	false	4-12-46	73
0159874645	21000	single	true	13-8-52	67
0218945958	35000	single	true	16-8-70	40

Next, let's look at what we should do with the features in our dataset.

Even if your data comes in a table, that doesn't necessarily mean that every column can be used as a feature right away (or that this would be a good approach). We'll need to look at the data provided and come up with a good way to translate it to a form in which the machine learning model is likely to learn from it. This depends both on what the data gives you, and on how your chosen model works.

Translating your raw data into features is more an art than a science, and the ultimate test is the [test set](#) performance. We'll look at a few simple examples, to give you an idea of the general way of thinking you should adopt.

**from:** date, phone number, images, status, text, category, tags, etc...

**to:** numeric, categoric, both.

Some algorithms (like linear models or kNN) work only on numeric features. Some work only on categorical features, and some can accept a mix of both (like decision trees).

## age

age	<b>to numeric:</b> From integer to real-valued. Not usually an issue.
41	
19	<b>to categoric:</b> Group data into bins? E.g. above or below the median.
28	<ul style="list-style-type: none"><li>Information loss is unavoidable.</li></ul>
25	
24	
73	
67	

This particular age column is integer valued, while numeric features are usually real-valued. In this case, we can just interpret the age as a real-valued number, and most algorithms won't be affected.

If our algorithm only accepts categoric features, we'll have to group the data into bins. For instance, you can turn the data into a two-category feature with the categories "below the median" and "above the median".

We'll lose information this way, which is unavoidable, but if you have a classifier that only consumes categorical features which works really well on the rest of your features, it may be worth it.

phone number	
phone nr	to numeric: From integer (?) to real-valued. Highly problematic.
0646785910	to categoric: area codes, cell phone vs. landline
0207859461	
0218945958	
0645789384	
0652438904	
0309897969	
0159874645	

We can represent phone numbers as integers too, so you might think a direct translation to numeric values is fine. But here it makes no sense at all. Translating a phone number to a real value would impose an *ordering* on the phone numbers that would be totally meaningless. My phone number may represent a higher number than yours, but that has no bearing on any possible target value.

What *is* potentially useful information, is the area code. This tells us where a person lives, which gives an indication of their age, their political leanings, their income, etc. Whether or not the phone number is for a mobile or a landline may also be useful. But these are **categorical features**.

Often in such cases, a single column in your raw data can yield several features for your machine learning model. For instance the phone number can give us area codes, but we can also derive from that whether the person lives in a big city or in the country side, whether the person has a cell phone or a landline, which province the person lives in, whether the person has a phone or not, etc. We could even extract a rough latitude/longitude in the form of two numeric features.

Some of these require a little work and creativity, but they can be extremely informative features. Especially compared to the raw phone number interpreted as an integer.

categoric to numeric	
integer coding:	one-hot coding:
genre	genre
sci-fi	1
romance	2
comedy	4
thriller	3
thriller	3
romance	2
romance	2
sci-fi	1
thriller	3
comedy	4

scifi   romance   comedy   thriller

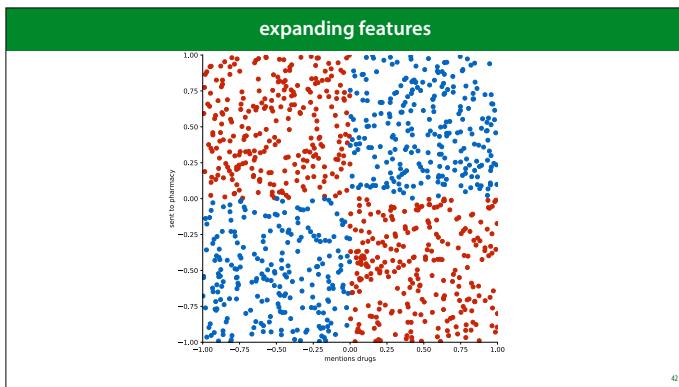
aka 1-of-N coding

So what if our model only accepts numerical features? This is very common: most modern machine learning algorithms are purely numeric. How do we feed it categorical data? Here are two approaches.

**Integer coding** gives us the same problem we had earlier. We are imposing a false ordering on unordered data.

**One-hot coding** (also called one-of-N coding) avoids this issue, by turning *one* categorical feature into *several* numeric features. Per genre we can say whether it applies to the instance or not.

In general, the one-hot coding approach is preferable. For almost all models, adding extra features does not substantially affect the runtime, and separating the different classes like this allows most models to use the information much more effectively.



Once we've turned all our features into data that our model can handle, we can still manipulate the data further, to improve performance.

How to get the useful information from your data into your classifier depends entirely on what your classifier can handle. The linear classifier is a good example. It's quite limited in what kinds of relations it can represent.

Essentially, each feature can only influence the classification boundary in a simple way. It can push it up or down, but it can't let its influence depend on the values of the other features. Here is a (slightly contrived) example of when that might be necessary.

Imagine classifying spam emails on two features: to what extent the email mentions drugs, and to what extent the email is sent to a pharmaceutical company. We'll consider the simplified case where every email that mentions drugs is spam, and every email that does not mention drugs is ham, unless the email is sent to a pharmaceutical company, in which case the roles are reversed.

With these two features, and this logic, we get the decision boundary shown here. This problem, called the **XOR problem** after the Boolean relation which produces the same picture, is completely impossible for a linear classifier to solve.

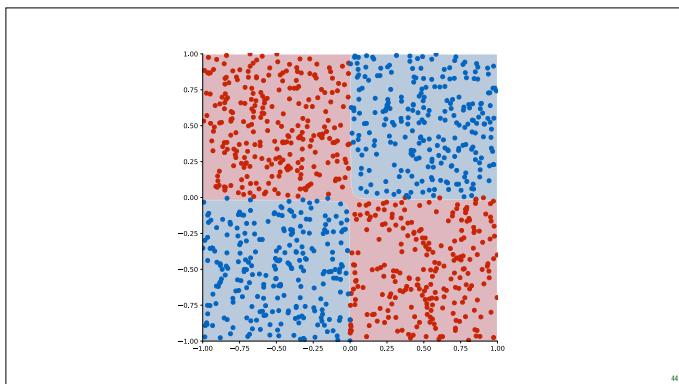
**cross product**

d	p	d * p
0.75	0.98	0.74
-0.66	-0.32	0.21
-0.45	0.84	-0.38
0.93	0.78	0.72
-0.42	0.24	-0.10
-0.02	0.43	-0.01
-0.74	0.58	-0.43
-0.41	-0.41	0.17
0.50	0.72	0.42

We can switch to a more powerful model, but we can also add power to the linear classifier by **adding extra features derived from the existing features**.

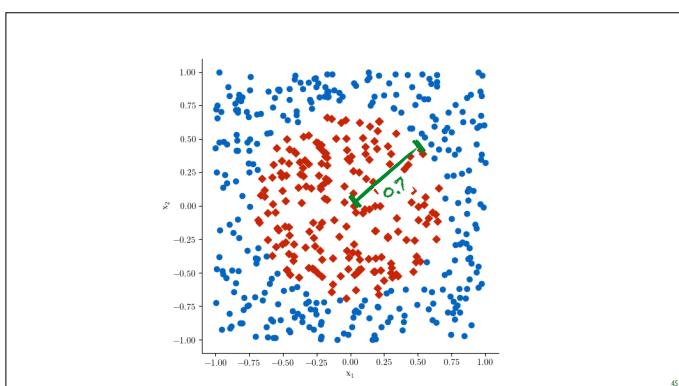
Here, we've added the cross-product of d and p (one value multiplied by the other). Note the XOR relationship of the signs: two negatives or two positives both make positive, a negative and a positive make a negative.

If we feed this three-feature dataset to a linear classifier, it can easily solve the problem. All it needs to do is to look at the sign of the third feature, and ignore the rest. We still have a simple linear classifier, which can easily and efficiently be optimally solved, but now it can learn non-linear relations in the original 2D feature space.



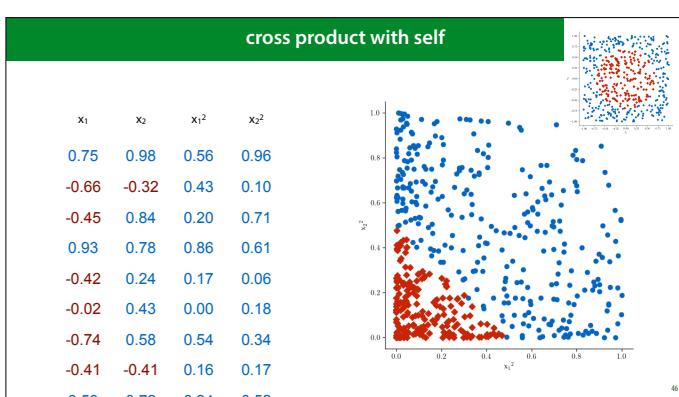
Here's what the result looks like for our data.

This is a linear classifier that operates in a 3D space. But since the third dimension is derived from the other two, we can colour our original 2D space with the classifications. Projected down to 2D like this, the classifier solves our XOR problem perfectly.



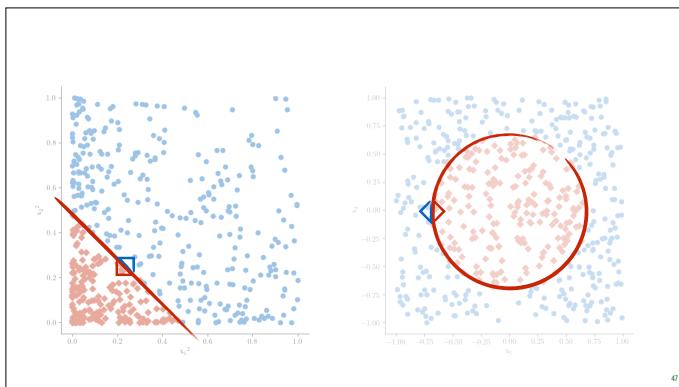
Here's one more example. In this dataset points are given the **negative class** if the distance to the origin is less than 0.7. Again, this problem is not at all linearly separable.

Using Pythagoras, however, we can express how the classes are decided: if  $x_1^2 + x_2^2 < 0.7^2$  then we classify as **red**, otherwise as **blue**. This is a linear decision boundary for the single feature  $x_1^2 + x_2^2$  or for the two features  $x_1^2$  and  $x_2^2$ .



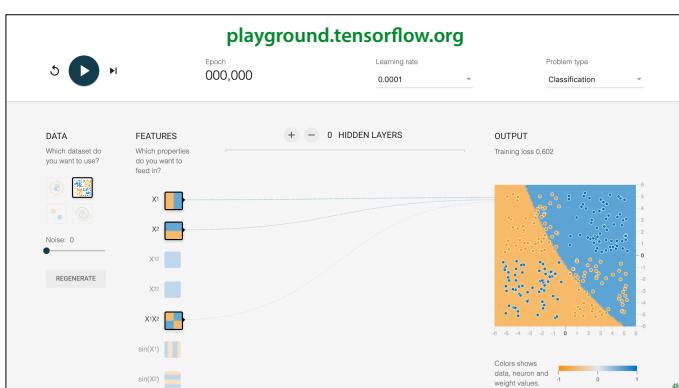
Here is how that looks in the second case. We add the two features  $x_1^2$  and  $x_2^2$  to the dataset, making it a 4-dimensional problem.

If we scatterplot just these two new axes, we see that the problem becomes linear. This is because deciding whether the sum of two features is larger than a particular value, is a linear problem.



So, if we add those features to the data, creating a 4D dataset, a linear decision boundary in that space solves our problem perfectly. This classifier can just ignore the two other features, and make its judgment purely on the values of the two new features we added.

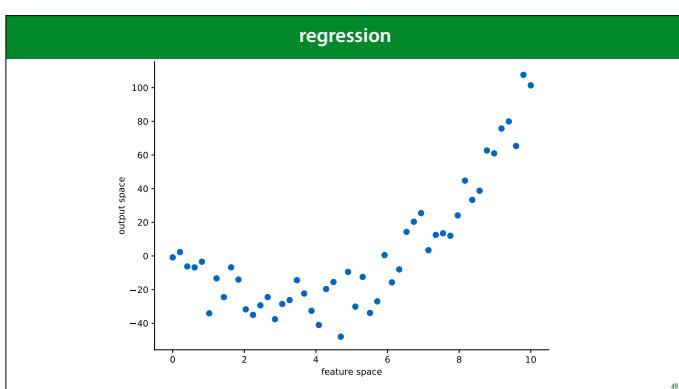
If we draw the decision boundary in the original space, we see that it forms a perfect circle.



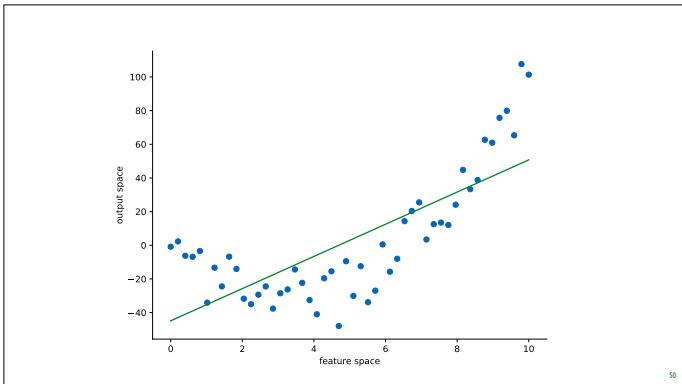
You can try this yourself at [playground.tensorflow.org](https://playground.tensorflow.org).

The column labeled features contains some extra features derived from the original two by various functions (including the cross product).

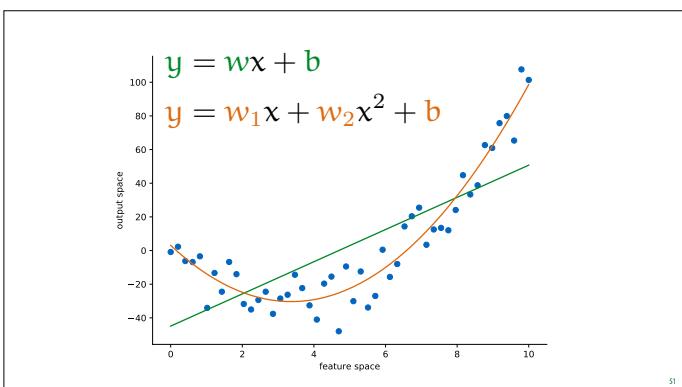
Note that both the XOR and Circle dataset are present.



We can do the same thing with regression. Here, we have a very non-linear relation.



A purely linear classifier does a terrible job.



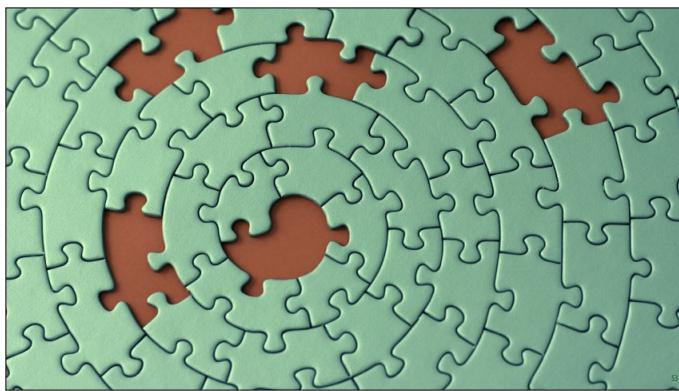
We can fit a **parabola** through the data much more closely. We can see this as a more powerful model (a parabola instead of a linear model), but we can also see this as a 2D linear regression problem, where the second feature ( $x^2$ ) is derived from the first.

This is relevant because linear models are extremely simple to fit. By adding derived features we can have our cake and eat it too. A *simple* model that we can fit quickly and accurately, and a *powerful* model that can fit many nonlinear aspects of the data.

If we don't have any intuition for which extra features might be worth adding, **we can just add all cross products** of all features with each other and with themselves (like  $x^2$ ). Other functions like the sine or the logarithm may also help a lot.

adding features
Can make a weak classifier (especially a linear one) stronger.
Any function of one or more of the existing features can work.
The problem stays convex: easy to solve, optimal solution guaranteed.
Common choice: all 2-way cross products, all 3-way cross products, etc.
$x, y, z \rightarrow x, y, z, xy, yz, xz, x^2, y^2, z^2, xyz, x^2y, \dots, x^3, \dots$

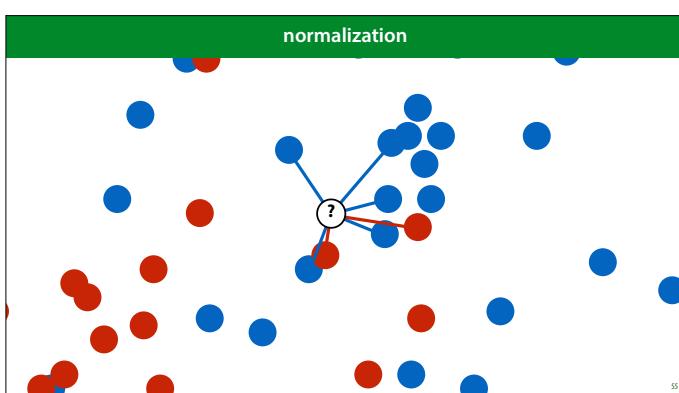
Adding all 2 and 3 way cross products blows up very quickly, but it's important to note that linear classifiers are extremely cheap to fit. We can still do it very quickly and efficiently, even if we have 100 000 features or more.



Data pre-processing  
Part 3: Normalization

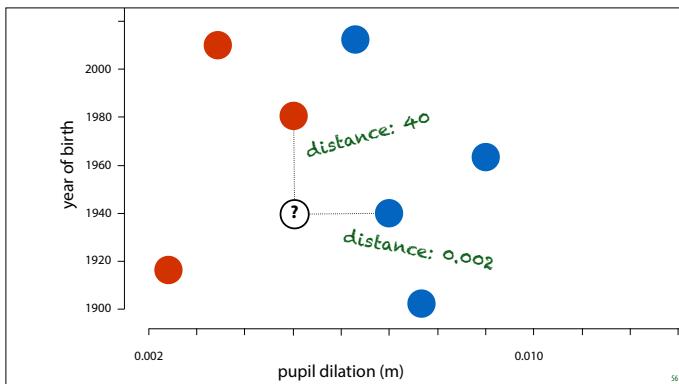
Machine Learning  
[mlvu.github.io](https://mlvu.github.io)  
Vrije Universiteit Amsterdam

|section|Normalization|  
|video|<https://surfdrive.surf.nl/files/index.php/s/xfpoE72AzWFlxPy/download>|



For some models, it's important to make sure that all numeric features have broadly the same minimum and maximum. In other words, that they are *normalized*.

To see why, let's revisit the k nearest neighbors (kNN) classifier from the first lecture.



Imagine we are using a 1-NN classifier (i.e. it only looks at the nearest example, and copies its class).

In this plot, it looks like the blue and the red dot are the same distance away.

But note the range of values for the two features: years and pupil dilation. Because years are measured in bigger units than pupils, the blue dot will always be much closer. But this distinction is not meaningful: we cannot compare durations to distances. The only thing that really matters is how close the point is to our target compared to the other points in the data. The absolute distance in the natural units doesn't matter.

What we want to look at is how much spread there is *in the data*, and use that as our distance. We do that by **normalizing** our data before feeding it to the model.

### creating a uniform scale

normalization  
fit to [0,1]

standardization  
fit to 1D standard normal distribution

whitening  
fit to multivariate standard normal distribution

57

We'll discuss three approaches to solving this problem.

**Normalization**, which reshapes all values to lie within the range [0, 1], **standardization**, which reshapes the data so that its mean and variance are those of a standard normal distribution (0 and 1 respectively) and **whitening**, which looks at features *together*, to make sure that as a whole their statistics are those of a multivariate standard normal distribution.

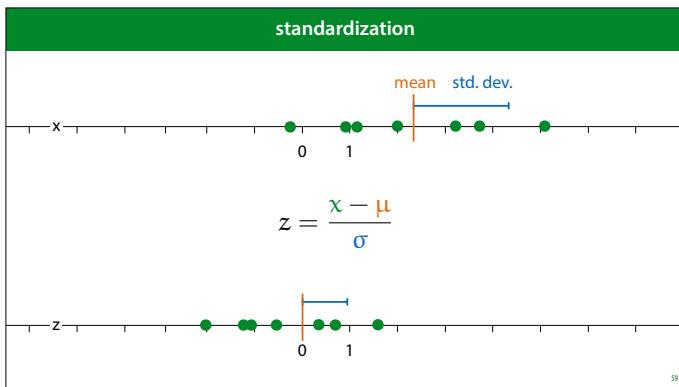
These terms are often used interchangeably. We'll stick to these definitions for this course, but in other contexts you should check that they mean what you think they mean.

### normalization

58

Normalization scales the data linearly so that the smallest point becomes 0 and the largest becomes 1. Note that because  $x_{\min}$  is negative (in this example), we are actually moving all data to the right, and then rescaling it.  $z$  is the normalized version of our data.

We do this independently for each feature.



Another option is standardization. We rescale the data so that the **mean** becomes zero, and the **standard deviation** becomes 1. In essence, we are transforming our data so that it looks like it was sampled from a standard normal distribution (or as much as we can with a one dimensional linear transformation).

"generating" the data	reverse
Sample $z$ from $N(0, 1)$	estimate $\mu$ and $\sigma$
translate to $N(\mu, \sigma)$ : $x = z\sigma + \mu$	translate back to $N(0, 1)$ : $z = (x - \mu)/\sigma$

60

The standardization operation is pretty simple, and maybe you can see where it comes from intuitively (it's pretty similar to the normalization operation), but even so, it's good to derive it carefully. This will prepare us for whitening, where we will do the same thing across multiple features.

For a rigorous derivation, we can think of the data as being "generated" from a standard normal distribution, followed by multiplication by  $\sigma$ , and adding  $\mu$ . The result is the distribution of the data that we observed. You can think of all normally distributed data being generated this way: sampled from a standard normal distribution, and scaled and translated to fit some non-standard distribution.

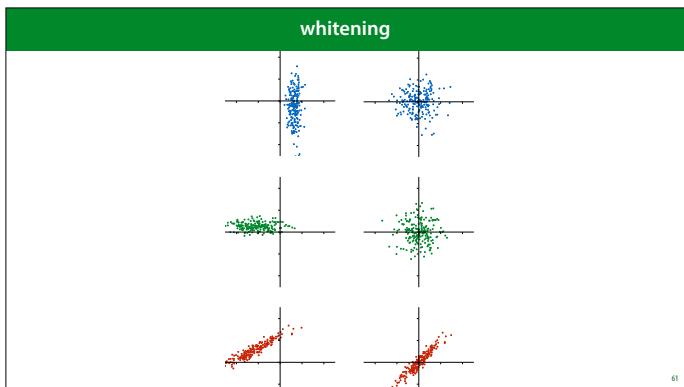
*If the data isn't normally distributed, we simply pretend that it is, and usually the results will still be fine. This is a common trend in machine learning, we care more about whether our assumptions work, than whether they are true.*

If we then compute the mean and the standard deviation of the data, the formula in the slide is essentially **inverting the transformation**. We reverse the order, and replace addition by subtraction and multiplication by division. This takes the distribution that we observed and recovers the "original" data as sampled from the standard normal distribution.

We will build on this perspective several times throughout the course.

Of course, in practice our data may not be normally distributed at all (standard or otherwise), so we should be a bit careful with these kinds of operations that assume a normal distribution. Still, if the data is roughly equally distributed over a finite range, without any extreme outliers, standardization will work fine for most models. And, if it really fails, then normalization will probably fail too, and you'll need to think about trying more exotic approaches, or even designing your own.

Remember, the proof is in the pudding: if the validation error is low, you probably did alright.



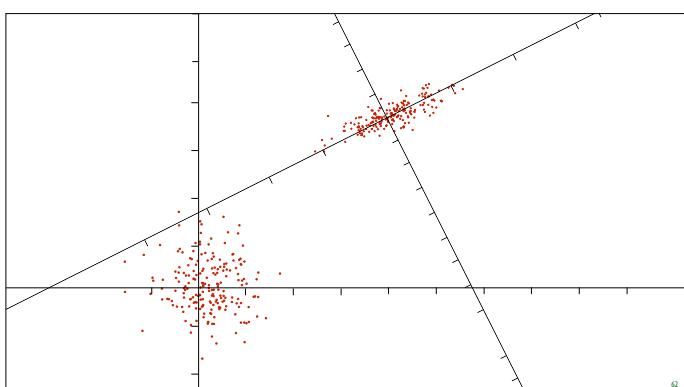
Here's what standardization looks like if we apply it to data with two features. If the data is *uncorrelated*, we are reducing it to a nice spherical distribution, centered on the origin, with the same variance in each direction. Exactly what data from a **multivariate standard normal distribution** looks like.

If, however, our data is *correlated*, that is; knowing the value of one feature helps us predict the value of the other, we get a different result. This is because we standardize each feature *independently*, and the features are not independent. Is there a way to achieve the same effect with the correlated data? Can we transform the features somehow so that it looks like they came from a distribution like the one top right? This is what **whitening** can do for us.

**Note that this is not usually necessary in practice.**

**Normalizing or standardizing each feature**

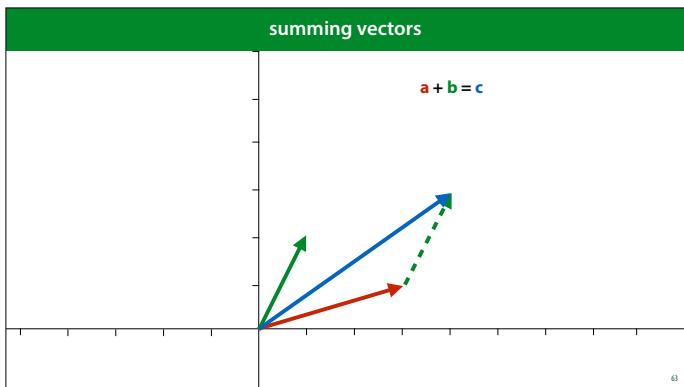
**independently is usually fine**, especially if your model is powerful enough to learn correlations. However, whitening, normalizing across features, can sometimes give you a little boost. It will also help us understand the PCA method, which we will discuss in the next video.



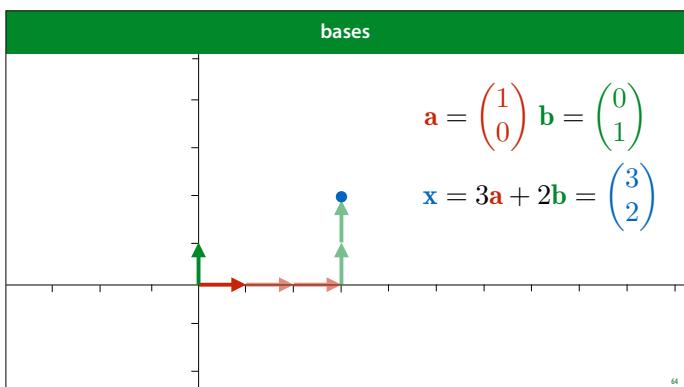
In essence we want to transform the data top right to something that looks like the data bottom left. Or, the same question asked differently, can we express the data in another **coordinate system**, to that in the new coordinate system, the features are not correlated and the variance in the direction of each axis is 1?

In order to show how to do this we need to revise some bits of linear algebra. Specifically, we need to look at **linear bases** (the plural of basis).

*We'll go through it a bit quickly, because we assume that you've already covered basis transformations in linear algebra. If not, or if your memory of them is hazy, you should take some time to review them.*

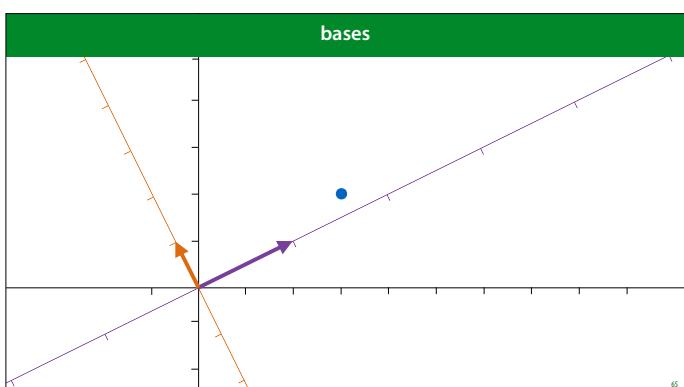


First, a quick reminder of how summing vectors works. We stick the tail of vector **b** onto the head of vector **a** and draw a line from the tail of **a** to the head of **b**. The point where we end up is the tip of the vector **a + b**.



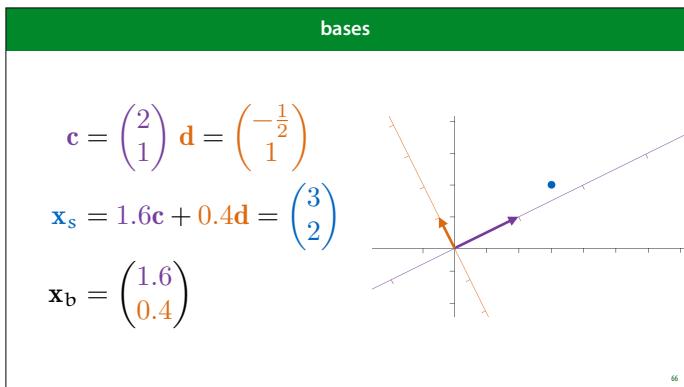
We can see our basic Cartesian coordinate system as made up entirely of the two vectors  $(1 \ 0)$  and  $(0 \ 1)$ . To describe a point in the plane, we just sum a number of copies of these vectors.

Every point in the plane is just a linear combination of these two. A coordinate like  $(3, 2)$  means: "sum three copies of **a** and add them to two copies of **b**." We call these **basis vectors**: vectors that allow us to describe all points in a space in terms of a multiple of each of the basis vectors. The set of points that can be described in this way is the space **spanned** by the basis vectors.

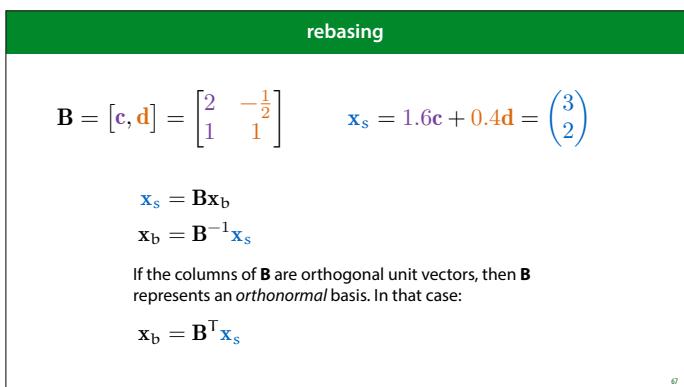


If we choose *different basis vectors*, we get a different **coordinate system** to express our data in. But (excepting some rare choice of basis vectors), we can still express all the same points as a number of copies of **one vector**, plus a number of copies of **the other**.

The blue point, which was at coordinates  $(3, 2)$  in our standard basis, is at coordinates  $(1.6, 0.4)$  in this new basis.



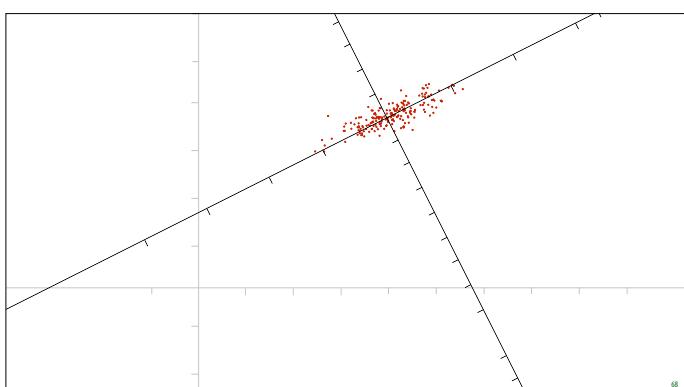
If we know the coordinates  $\mathbf{x}_b$  in our non-standard coordinate system, it's easy to find the coordinates  $\mathbf{x}_s$  in the standard basis. We just multiply the first coordinate of  $\mathbf{x}_b$  with the first basis vector, the second coordinate with the second basis vector and sum the result.



The basis vectors are usually expressed as the columns of a matrix  $\mathbf{B}$ . That way, transforming a coordinate  $\mathbf{x}$  in basis  $\mathbf{B}$  to the standard coordinates can be done simply by matrix multiplying  $\mathbf{B}$  by  $\mathbf{x}$ . It also tells us that to go the other way, to transform a standard coordinate to the basis, you multiply by the *inverse* of  $\mathbf{B}$ .

Since inverting a matrix is an expensive and numerically unstable business, it's good to focus (if possible) on **orthonormal bases**. That is, bases for which the basis vectors are **orthogonal** (the angle between any two of them is 90 degrees) and **normal** (all vectors have length 1). In that case the matrix transpose (which is simple to compute without loss of precision) is equal to the matrix inverse, so we can switch back and forth between bases quickly, without losing information.

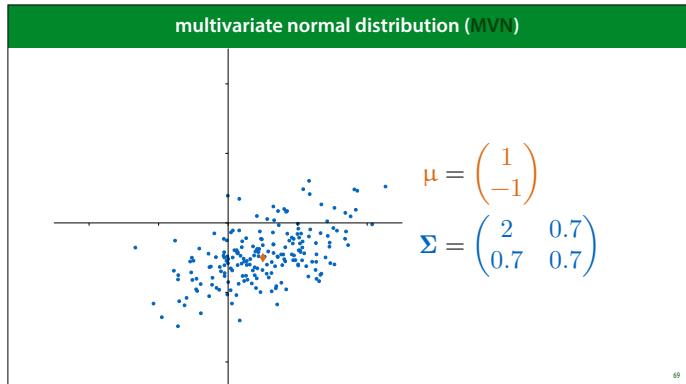
Here  $[\mathbf{a}, \mathbf{b}]$  represents the matrix created by concatenating the vectors  $\mathbf{a}$  and  $\mathbf{b}$ .



We can now re-phrase what we're aiming to do: we want to find a set of new *basis vectors* so that we can express the data in a coordinate system where the features are not correlated, and the variance is 1 in every direction.

Note that the latter means we can't have an orthonormal basis (the basis vectors can't be one).

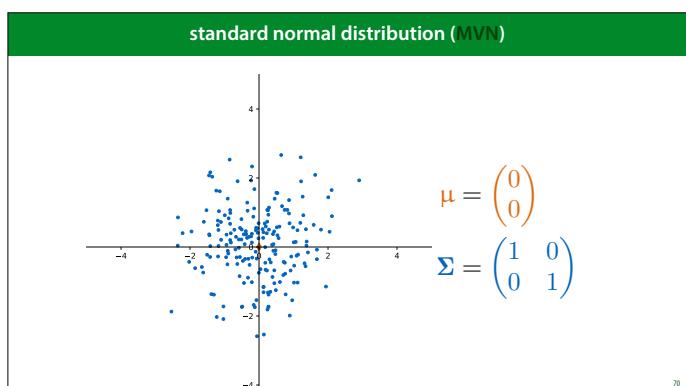
*We can fix this by first computing an orthonormal basis, and then scaling independently along each axis, but we won't go into that here. For now, we'll just allow for non-orthonormal bases.*



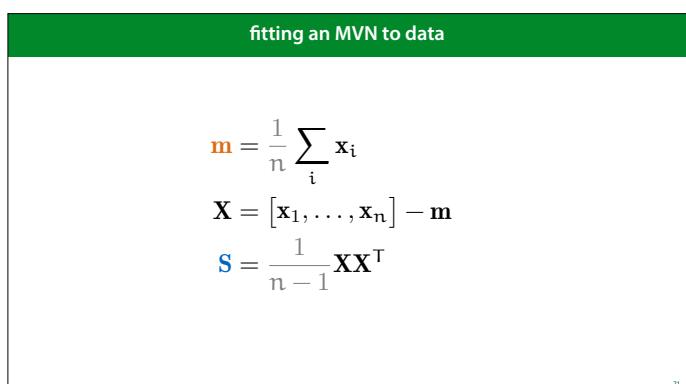
To figure out how to find this basis, we will follow the same principle as we did with standardisation: we will assume that the data was generated by a standard multivariate normal distribution (MVN), followed by a translation and a change of basis (with the change of basis causing some features to become correlated). We will attempt to reverse the process by:

- fitting a (nonstandard) MVN to the data
- figuring out the transformation that transforms the standard MVN to this MVN
- applying the inverse of this transformation to the observed data

A multivariate normal distribution is a generalisation of a one-dimensional normal distribution. Its mean is a single point, and its variance is determined by a symmetric matrix called a **covariance matrix**. The values on the diagonal indicate how much variance there is along each dimension. The off-diagonal elements indicate how much co-variance



The *standard* MVN has its mean at the origin and the identity matrix as its covariance matrix (i.e. its features are uncorrelated, and the variance is 1 along every dimension).



The maximum likelihood estimators for the **sample mean**  $\mathbf{m}$  and **sample covariance**  $\mathbf{S}$  look like this. Computing these values lets you fit an MVN to your data.

You can derive these by starting with the probability density function we saw in the last slide, applying the maximum likelihood objective, setting the derivative equal to zero and solving for the mean and for the covariance. You don't have to be able to do this (it gets a bit technical), but you do have to understand the idea.

### MVNs as transformations

Start with  $N(\mathbf{0}, \mathbf{I})$  ← standard normal

- Let  $Z \sim N(\mathbf{0}, \mathbf{I})$
- Let  $X = \mathbf{A}Z + \mathbf{t}$

then:

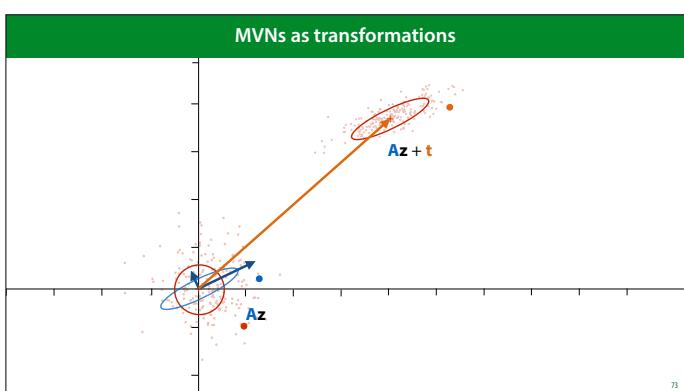
$$X \sim N(\mathbf{t}, \mathbf{A}\mathbf{A}^T)$$

As before, we will imagine that our data originally came from a standard MVN, and was then transformed to the data we observed by multiplying each point by some matrix  $\mathbf{A}$  (changing the basis) and then adding some vector  $\mathbf{t}$  (moving the mean away from the origin).

*We can sample a point  $\mathbf{z}$  from an  $n$ -dimensional standard MVN by simply filling  $\mathbf{z}$  with values sampled from a one-dimensional standard normal distribution.*

If we then transform  $\mathbf{z}$  by multiplying it by some matrix  $\mathbf{A}$  and adding some vector  $\mathbf{t}$ , the result is the same as sampling from an MVN with mean  $\mathbf{t}$  and covariance  $\mathbf{A}\mathbf{A}^T$ .

Any MVN can be described in this way as a transformation of the standard normal distribution.



Here's what that looks like. For our data. We imagine some data sampled from a standard MVN. We multiply by some matrix  $\mathbf{A}$  to squish and rotate it. And then we apply a translation vector  $\mathbf{t}$  to translate it to the right point in space.

### whitening: invert this transformation

- Compute  $\mathbf{S}, \mathbf{m}$  from data.
- Find some  $\mathbf{A}$  such that  $\mathbf{S} = \mathbf{A}\mathbf{A}^T$ 
  - Cholesky decomposition
  - Singular Value Decomposition (PCA whitening)
  - Matrix square root
- Whiten the data:

$$\mathbf{z} = \mathbf{A}^{-1}(\mathbf{x} - \mathbf{m})$$

Now, we need to invert this. Given some data, we fit an MVN, find out which  $\mathbf{A}$  and  $\mathbf{t}$  match that MVN, and then invert the transformation from the standard MVN to the observed data.

In slide 72, we saw that the covariance after our transformation was  $\mathbf{A}\mathbf{A}^T$ , so if we estimate the covariance  $\mathbf{S}$  and find some matrix  $\mathbf{A}$  such that  $\mathbf{A}\mathbf{A}^T = \mathbf{S}$ , we can then use that  $\mathbf{A}$  for the inverse transformation.

*Finding this  $\mathbf{A}$  can be done in many ways. The most stable and popular one is the singular value decomposition (SVD), which leads to a method known as PCA whitening, discussed in the next video.*

Since the multiplication by  $\mathbf{A}$  doesn't change the mean, we know that the translation vector  $\mathbf{t}$  is equal to the mean  $\mathbf{m}$ .

Once we know  $\mathbf{A}$  and  $\mathbf{t}$ , we can reverse the transformation as shown here.

Compare this to the standardization operation we saw earlier: there, we subtract the mean, and multiply by the

inverse of the standard deviation. Here we do the same, but in multiple dimensions

*Note that the standard deviation is the square root of the variance, just like the  $A$  matrix squared is the covariance.*

## recap

Rescale your data so that the *relative* differences between instances are emphasized.

**Normalization** or **standardization** are usually good enough.

**Whitening** is a more powerful method. Usually not necessary, but it pays to understand it.

75

|section|Principal Component Analysis  
|video|<https://surfdrive.surf.nl/files/index.php/s/G5jhEGFlwdRgvOw/download>|

## Data pre-processing

Part 4: Principal Component Analysis

Machine Learning

[mlv.u.github.io](https://mlv.u.github.io)

Vrije Universiteit Amsterdam

## dealing with too many features

Feature selection:

Select a *subset* of the existing features to operate on.

**Dimensionality reduction:**

Map the features to a new *smaller* set of features. Retain as much information as possible.

Some datasets have more features than a given model can handle. Or, maybe the model can handle it, but it's overfitting on all the noise that so many features introduce.

In that case, there are two things we can do: we can try to find a subset of the features that is most informative, and operate on those. This has the benefit that the features retain their meaning and are still interpretable. This is called **feature selection**.

The alternative is to take information from all *all* features and to map them to a new (smaller) set of derived features, which retain as much of the original information as possible. This is called **dimensionality reduction**. In this case, the new features don't always have an obvious meaning, but they may still work well for machine learning purposes.

In this video we will just look at one dimensionality reduction method: Principal Component Analysis (PCA). We won't discuss feature selection, but if you're interested, a good place to start is the methods that come with sklearn:  
[https://scikit-learn.org/stable/modules/feature\\_selection.html](https://scikit-learn.org/stable/modules/feature_selection.html)

## dimensionality reduction

	data						
1	.3	.2	.3	...	.0	.6	2
2	.1	.4	.0	...	.3	.0	9
3	.6	.4	.0	...	.0	.3	5
4	.6	.0	.1	...	.0	.4	5
5	.5	.0	.3	...	.1	.3	6
6	.3	.3	.1	...	.3	.1	8
7	.2	.0	.2	...	.4	.0	2
8	.3	.8	.4	...	.3	.0	3
9	.9	.6	.6	...	.1	.0	4
10	.0	.9	.3	...	.3	.1	4
11	.0	.6	.0	...	.6	.1	0
12	.6	.0	.3	...	.0	.2	0
13	.3	.6	.6	...	.0	.6	3
14	.6	.0	.7	...	.0	.7	0

28x28 pixels = 784 features

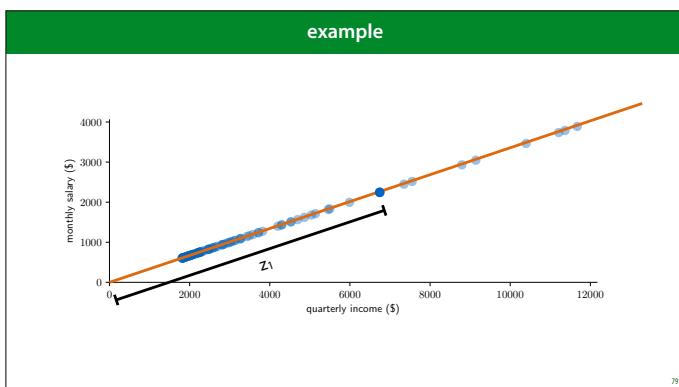
	$z_1$	$z_2$	$z_3$	
1	1.5	.2	-.3	2
2	3.4	.4	.0	9
3	-2	.5	-.2	5
4	-.1	.0	.1	5
5	-2.1	.0	.3	6
6	-.8	.3	-.1	8
7	-.1	.0	.2	2
8	.0	-.8	.4	3
9	4	-.6	-.6	4
10	-2	.9	.3	4
11	0	-.6	.0	0
12	-.7	-.5	.3	0
13	3	-.6	.3	3
14	-1	.1	.3	0

Dimensionality reduction is the opposite of the feature expansion trick we saw earlier. It describes methods that reduce the number of features in our data (the dimension) by deriving new features from the old ones, hopefully in such a way that we capture all the essential information. There are several reasons you might want to reduce the dimensionality of your data:

- **Efficiency.** Some machine learning methods can only handle so many features. If you have a very high dimensional dataset, you may be forced to do some dimensionality reduction in order to be able to run your chosen model.
- **Reduce variance** of the model performance (make the bias/variance tradeoff). Feature expansion boosts the power of your model, likely giving it higher variance and lower bias. Dimensionality reduction does the opposite: it reduces the power of your model likely giving you higher bias and lower variance.
- **Visualization.** If you're lucky (or if you have a very

strong dimensionality reduction method), reducing down to just 2 or 3 dimensions preserves the important information in your data. If so, you can do a scatterplot, and use the power of your visual cortex to analyse your data (i.e. you can look at it).

*We'll use  $x$  for the original features and  $z$  for the reduced features. This is the same letter we used in normalization, but as we will see later, normalization and dimensionality reduction have a lot in common.*

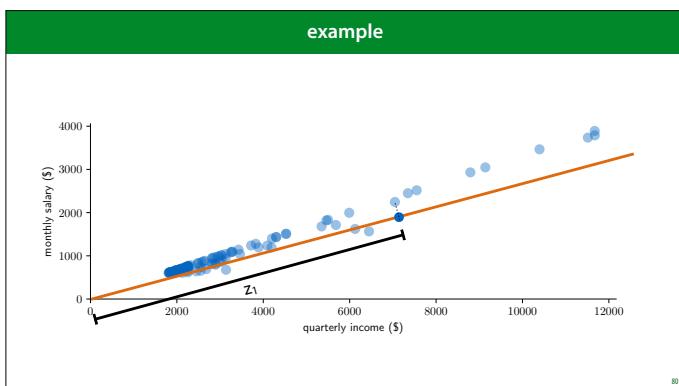


Dimensionality reduction makes most sense when there are *correlations* between your features. That is, when you can predict the value of one feature from the value of another with some success.

Here is a very simple example: imagine a dataset of people's income that contains their income per quarter and their salary per month. In this case, one of the features is entirely redundant. This is apparent from the fact the data forms an exact line when we plot it.

We can just store the quarterly income and multiply it by 3, or store the quarterly income and divide it by 3.

Or, we could **draw a line through the data, and describe each instance by how far far along the line it is**. So long as we know what the line is, this **one number**  $z_1$  is enough to perfectly reconstruct both features. This is the basic idea from which we will develop principal component analysis.



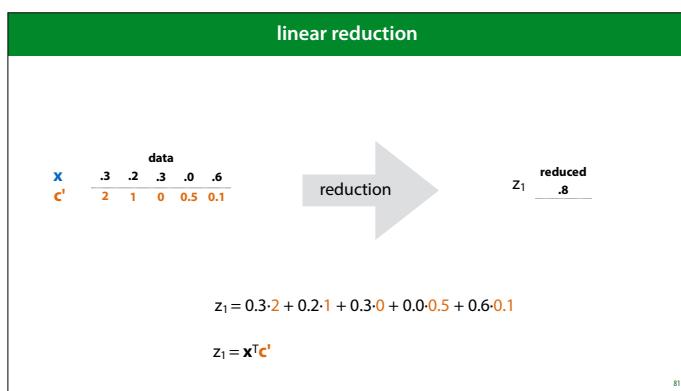
In practice, it's quite rare that our data is so perfectly correlated. But there is often *some* correlation.

Imagine, for instance, that we take into account that people have other sources of income beyond their salary (perhaps bonuses, one-time jobs, or investments paying off). In that case, the salary will still be the most important source of income, but we can expect the quarterly income to be a little different from three times the salary.

The data still lies roughly on a line, but no longer perfectly. We can't draw a perfect line through all points, but we can fit a line roughly through them. Then, we can represent each point by **projecting it onto the line** and measuring how far along the line the point is. This time the number  $z_1$  won't be enough to perfectly reconstruct both features, we can only reconstruct points that are exactly on the line, but if the line is a good fit through the points, our reconstruction will be pretty close.

Here,  **$z$  is the reduced feature**. If we find a well-fitting line, we can use the one-dimensional feature  $z$  instead of the two

features  $x_1$  and  $x_2$  and hopefully still retain enough information for machine learning algorithms to work well on this reduced data.

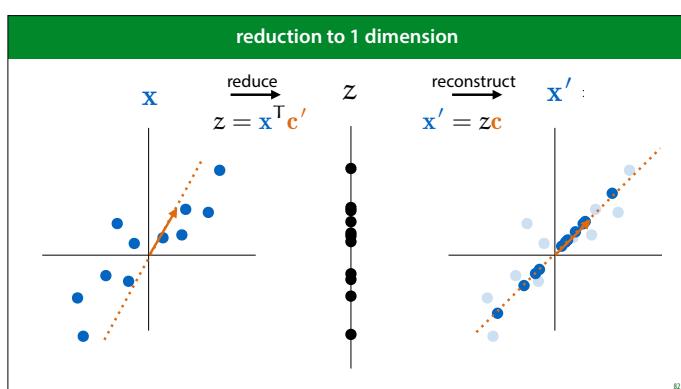


We will restrict ourselves to **linear reductions**. To create one of the derived features  $z_1$ , the only thing we are allowed to do is to pick **one number** for each feature, multiply them together and sum the result. These values we multiply by the original features should be the same for all instances.

If we arrange these multipliers in a vector  $\mathbf{c}'$  then we can simply say that the reduced feature is the dot product of the original features  $\mathbf{x}$  and the parameter vector  $\mathbf{c}'$ . The vector  $\mathbf{c}'$  determines the direction of the line in the previous slide.

If we want more than one reduced feature, we can add additional parameter vectors. However, to keep things simple, we start with just one.

The question is, how do we choose the elements of  $\mathbf{c}'$ ?



Since we're focusing on a single feature for now, we'll drop the subscript and call this feature "z". This is a single scalar value representing our entire instance  $\mathbf{x}$ .

The way we'll find the parameters  $\mathbf{c}'$  for our reduction is by optimizing the **reconstruction error**. We'll come up with some function that reconstructs our data from the reduced point  $z$ . The closer this reconstruction is to the original point, the better. It hopefully makes some intuitive sense that the better we can reconstruct  $\mathbf{x}$  from  $z$ , the more information from  $\mathbf{x}$  has been retained in  $z$ .

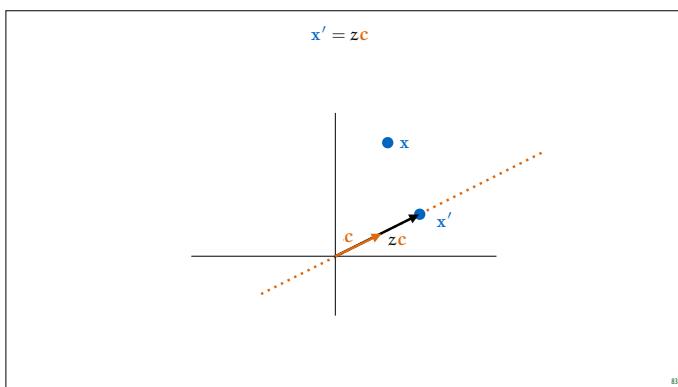
To keep things simple, *both* the function that reduces the data and the function that reconstructs the data should be **linear**. This means that our reconstruction is just some second vector  $\mathbf{c}$ , which we also get to choose, multiplied by the reduced feature  $z$ . We'll also assume that the data is **mean-centered**, so that we won't need to apply any translations: the mean of the original data, the reduced data, and the reconstructed data is zero or the zero vector.

*If the data isn't mean-centered, we just subtract the mean*

from each instance before we start the principal component analysis.

To recap, under these constraints, the reduction function consists of taking the dot product of our vector with some parameter vector  $\mathbf{c}'$ , and the reconstruction function consists of multiplying our reduced representation with some other parameter vector  $\mathbf{c}$ .

We will try to choose our parameters  $\mathbf{c}'$  and  $\mathbf{c}$  in such a way that  $\mathbf{x}$  is as close as possible to  $\mathbf{x}'$ . Before we figure out how to do this, however, we can simplify our problem. We can show that for the optimal solution, **the vectors  $\mathbf{c}'$  and  $\mathbf{c}$  must be the same**. We'll show that first.



Here is the reconstruction of  $\mathbf{x}$  from  $\mathbf{z}$  isolated in a diagram. Take a moment to study this picture. Note that we have fixed a line by our choice of  $\mathbf{c}$ , and our reconstruction, because it can only be a multiple of  $\mathbf{c}$ , must be somewhere on the line.

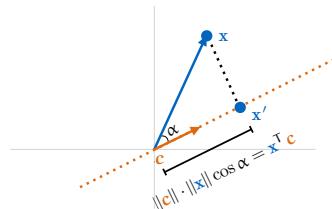
We'll work out what our functions should be in the following order. First, we will assume that we have the reconstruction function, and ask what the best reduction function is to use. Then we will work out an optimization objective for both of them together.

Imagine that  $\mathbf{c}$  is fixed. This could be at the optimal value, or some terrible value, but somebody has chosen  $\mathbf{c}$  for us and we're not allowed to change it. Which value should we choose for  $\mathbf{z}$  to put  $\mathbf{x}'$  as close to  $\mathbf{x}$  as possible?

Given  $\mathbf{c}$ , what's the closest we can get to  $\mathbf{x}$  while staying on the line? If you remember your linear algebra you'll know that this is the point where the line between  $\mathbf{x}$  and  $\mathbf{x}'$  makes a right angle with the line of  $\mathbf{c}$ . This is the **orthogonal projection** of  $\mathbf{x}$  onto  $\mathbf{c}$ . What you should also remember from linear algebra is that the length of  $\mathbf{z}\mathbf{c}$  in this picture is related to the dot product of  $\mathbf{x}$  and  $\mathbf{c}$ . Why?

### the dot product provides the orthogonal projection

$$\mathbf{x}' = z\mathbf{c}$$



assume that  $\mathbf{c}$  is a unit vector

From basic trigonometry, we know that the length of the black line is  $||\mathbf{x}|| \cos \alpha$ .

For our purposes, the length of  $\mathbf{c}$  doesn't matter (if we make  $\mathbf{c}$  longer or shorter it still defines the same line), so **we'll assume that it has length 1** (that is, it is a *unit vector*).

Because  $||\mathbf{c}|| = 1$ , we can multiply by that without changing the value, which means that the length of the black line is equal to  $||\mathbf{c}|| \cdot ||\mathbf{x}|| \cos \alpha$ , the dot product between  $\mathbf{x}$  and  $\mathbf{c}$ .

*If this seems a bit magical, see [peterbloem.nl/blog/pca](http://peterbloem.nl/blog/pca) (required reading) for a more intuitive proof. It all boils down to the Pythagorean theorem in the end.*

reduction

$$\begin{aligned} z &= \mathbf{x}^T \mathbf{c}' \\ z &= \mathbf{x}^T \mathbf{c} \end{aligned}$$

optimal reconstruction

$$z = \mathbf{x}^T \mathbf{c}$$

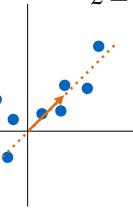
What this tells us, is that the orthogonal projection of  $\mathbf{x}$  onto  $\mathbf{c}$  is found by taking the dot product of  $\mathbf{x}$  and  $\mathbf{c}$ . Since  $\mathbf{c}$  has length one, this is the value that we want to multiply  $\mathbf{c}$  by to get to  $\mathbf{x}'$ .

When we started, we assumed that we had two parameter vectors:  $\mathbf{c}$  for the reconstruction, and  $\mathbf{c}'$  for the reduction. Now we find that whatever we choose for  $\mathbf{c}$ , setting  $\mathbf{c}' = \mathbf{c}$  provides the optimal value of  $z$ .

### reduction to 1 dimension

$\mathbf{x}$

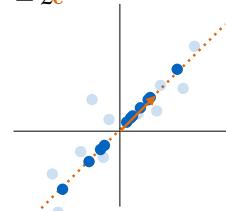
$$\xrightarrow{\text{reduce}} z = \mathbf{x}^T \mathbf{c}$$



$$z = \mathbf{x}^T \mathbf{c}$$

$$\xrightarrow{\text{reconstruct}} \mathbf{x}' = z\mathbf{c}$$

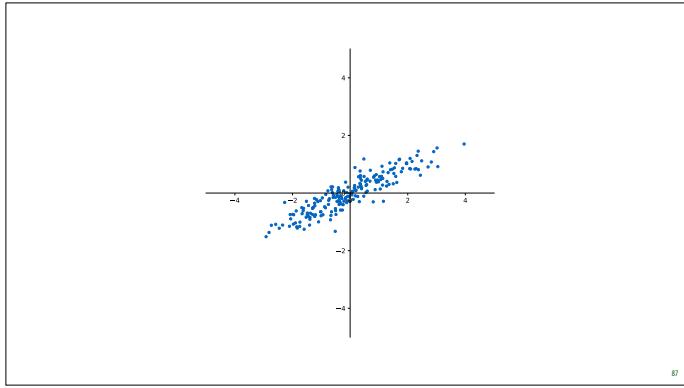
$$\mathbf{x}' = z\mathbf{c}$$



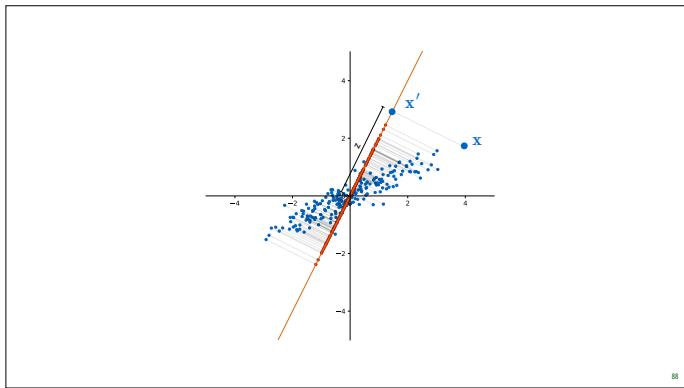
This is the simplified picture: the reduction and reconstruction now have the same parameters  $\mathbf{c}$ . Note that this required an additional assumption: that  $\mathbf{c}$  is a unit vector.

So here's the model: we pick some unit vector  $\mathbf{c}$ , project our data onto it to represent it as a single scalar  $z$ , and then, to reconstruct the data, multiply  $\mathbf{c}$  by  $z$ . As you can see, the reconstructed data necessarily lie on a line. All we are looking to do is to get these reconstructions as close to the original points as possible. If we manage that, it's reasonable to assume that we retained some information from the original features in the single reduced feature  $z$ .

The only remaining question is, which  $\mathbf{c}$  should we choose to minimize the reconstruction error?



Here's some randomly generated data. Remember, we assumed that the data would be mean centered. Let's first pick a random parameter  $\mathbf{c}$ . And see what we get.



Here it is. The red points are our reconstructions. For each point, the new feature  $z$  is the distance from the origin to the red point. The grey lines indicate how far the reconstruction is from the original data. Note that these grey lines are orthogonal to the line described by  $\mathbf{c}$ , because we are reducing our data by orthogonally projecting it onto  $\mathbf{c}$ .

Clearly, this is not a very good choice for  $\mathbf{c}$ . The grey lines could be much shorter. This is how we'll optimize for  $\mathbf{c}$ . **We'll sum up the squares of the grey lines and minimize that sum.**

We can think of optimizing  $\mathbf{c}$  as making the grey lines rubber bands, that pull on the line representing  $\mathbf{c}$  (which pivots around the origin).

This is a lot like linear regression, but the task is slightly different. Note that there is no target attribute here, and the "residuals" are not parallel to one of the axes.

*Doing regression with the residuals drawn like this is*

objective
$\arg \min_{\mathbf{c}} \sum_{\mathbf{x}} \ \mathbf{x}' - \mathbf{x}\ ^2$
$= \arg \min_{\mathbf{c}} \sum_{\mathbf{x}} \ z \cdot \mathbf{c} - \mathbf{x}\ ^2$
$= \arg \min_{\mathbf{c}} \sum_{\mathbf{x}} \ \mathbf{x}^T \mathbf{c} \cdot \mathbf{c} - \mathbf{x}\ ^2$
$= \arg \min_{\mathbf{c}} \sum_{\mathbf{x}} \sqrt{\sum_i (\mathbf{x}^T \mathbf{c} \cdot \mathbf{c} - \mathbf{x}_i)^2}$
$= \arg \min_{\mathbf{c}} \sum_{\mathbf{x}, i} (\mathbf{x}^T \mathbf{c} \cdot \mathbf{c} - \mathbf{x}_i)^2$
such that $\ \mathbf{c}\  = 1$

To find a better  $\mathbf{c}$ , we will simply state our goal as an optimization objective. We want to find the  $\mathbf{c}$  for which the squared distance between the data and the reconstructed data is minimized. We first fill in the definition of the reconstruction (line 2), and then the definition of the optimal  $z$  (line 3).

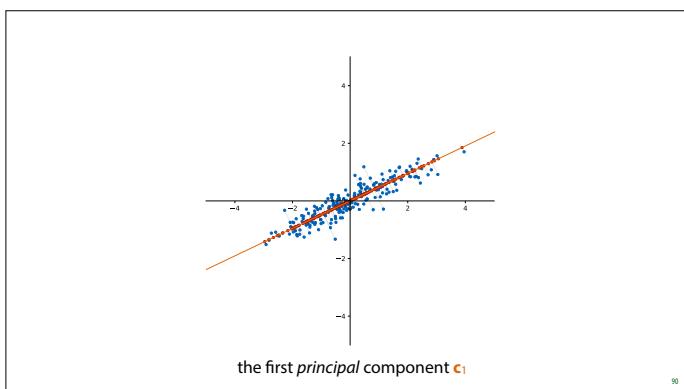
In the definition of the Euclidean distance, the square root cancels out against the square in our optimization, so that we are left with a sum of the squares over every dimension in every reconstructed instance  $\mathbf{x}'$ .

This leaves us with a simple objective to which we can apply any search algorithm, like gradient descent. One thing we must remember: we required that  $\mathbf{c}$  is a unit vector.

*Without this requirement we don't get  $\mathbf{c}' = \mathbf{c}$ .*

This means we have an optimization problem with a *constraint*. This is a technical subject (have a look at the extra lecture on SMVs if you're interested). For now, we can solve this problem with a simple trick: we apply gradient

descent and normalize the vector  $\mathbf{c}$  to scale it back to a unit vector after every gradient update. This is called the projection method for constrained optimization. It doesn't always work, but it does here. In practice, there are much more efficient ways of computing PCA anyway, we're just using gradient descent here because it's a method we're familiar with.



We run gradient descent and this is the solution that we find. It looks pretty good. It's hard to imagine any other line  $\mathbf{c}$  leading to shorter grey lines.

*Note that the data is much more spread out along this line than it was for our earlier choice of  $\mathbf{c}$ .*

We call this  $\mathbf{c}$  the **first principal component** of the data.

### for more dimensions

Repeat the process on the remaining directions.

The *second* principal component  $\mathbf{c}_2$  is the unit vector

- orthogonal to  $\mathbf{c}_1$
- which, together with  $\mathbf{c}_1$ , minimises the loss

Can be computed all in one go

Out of scope for this course

If we want to reduce the dimensionality to more than one dimension, we repeat the process. Keeping the first principal component fixed, the second principal component is the one orthogonal to the first that minimizes the reconstruction loss. This gives us two directions, orthogonal to one another, to project onto. Our reduced dataset then has two features.

Each next principal component is the direction orthogonal to all the previous ones, that minimizes the loss, when the data is reconstructed *using all of them*.

*This works well as a definition, but in practice we don't need to compute the principal components one by one. We just tell some algorithm to give us the first  $k$  principal components, and it spits them all out right away. How this works exactly, is out of scope for this course. The results are the same as solving this optimization problem one by one, just quicker.*

***The blog post in the required reading*** has some (non-required) follow ups that go into more detail about this.

## another perspective

The first principal component is the direction in which the **variance** is maximal.  
I.e. the projected data is most *spread out*.

The second principal component is the direction, orthogonal to the first, in which the remainder of the variance is maximal.

And so on.

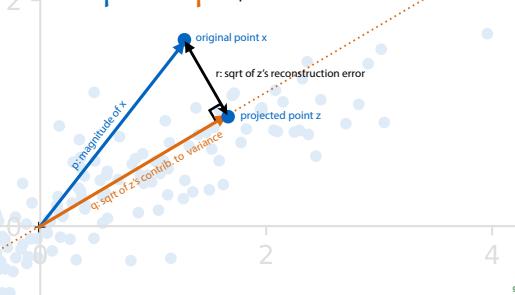
If you've heard about PCA before, you may be surprised by this definition using reconstruction loss. Usually, the principal components are defined as the directions in which the **variance** of the projected data is maximized. The best **c** is the line along which the orthogonal projections are the most spread out.

The first principal component is the line along which the variance of the data is maximal when projected onto the line. The second principal component is the line orthogonal to the first for which the variance is maximal, and so on.

It turns out, these two definitions are equivalent.

### least squares is variance maximisation

$$p^2 = q^2 + r^2$$



Let's look at the one dimensional reduction again to show why.

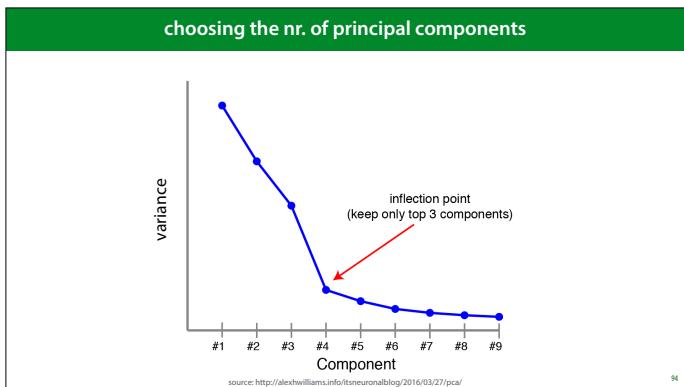
The variance of a one-dimensional dataset is defined as the average of the squares of all the distances to the data-mean. In our case, both the data and the reduction are mean-centered, so the variance is just the sum of all the squares of the z's; our reduced representations. In this picture, the length of the orange vector.

Thus, maximising the variance, means choosing **c** so that the (squared) length of the orange vector is maximized

This arrangement into a right-angled triangle means that the **magnitude of the original data** (**p**, the squared distance to the mean) is related to the **variance of the projected data** (**q**) and the reconstruction error (**r**, in black) by the Pythagorean theorem.

Since **p**, the magnitude of the original data, is a constant, **q**<sup>2</sup> + **r**<sup>2</sup> is constant, and minimizing the squared reconstruction error **r**<sup>2</sup> is equivalent to maximizing the variance of the projected data **q**<sup>2</sup>.

In the variance maximization view of PCA, we often talk about how much variance the reduced data *retains*, seeing the variance as a kind of "information content" in a representation of the data. A perfect reconstruction has the same total variance as the data.

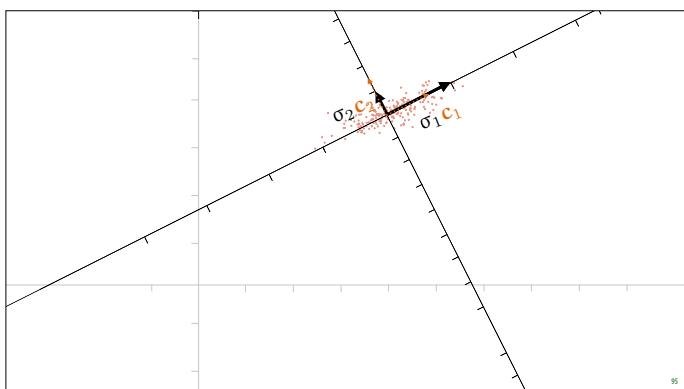


To use PCA for dimensionality reduction, we need to choose the number of dimensions to reduce to. We can just treat this as a hyperparameter and test different values.

But if we plot the variance or the reconstruction loss against the number of components, we often see a natural *inflection point*. In this case, we can retain the majority of the variance in the data by keeping only the first three principal components. The higher components still add a little variance each, but not much.

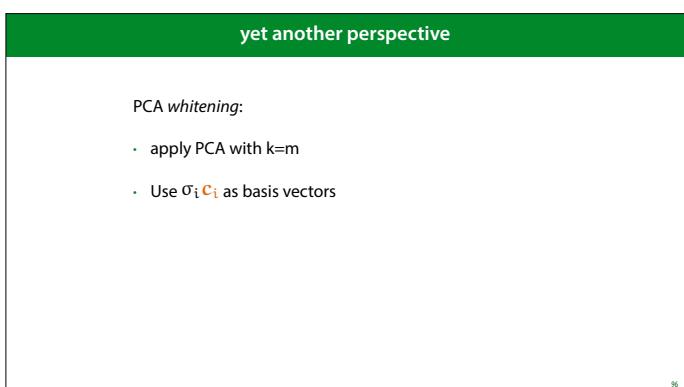
What happens if we keep going until the new data has the same number of features as the original?

source: <http://alexhwilliams.info/itsneuronalblog/2016/03/27/pca/>



If we do that, we get perfect reconstructions, but our  $z$ 's are still different from the original coordinates. We end up expressing the data in *another basis*, called the **PCA basis**. It turns out, that this actually gives us a **whitening** of the data: in the new basis, the data is uncorrelated, with variance 1 along each axis.

The different principal components  $c$  are unit vectors, which are by definition all mutually orthogonal. This means that the vectors  $c$  form an orthonormal basis. If we multiply each with the standard deviation of the data projected onto  $c$ , we end up with a whitening basis.



This way of whitening is called **PCA whitening**. We apply PCA with the same number of target dimensions as data dimensions. This gives us an orthonormal basis in which the data is uncorrelated. If we then measure the standard deviation along each component and multiply the basis vectors by that, we get a basis in which the data is whitened.

While  $\sigma$  and  $c$  together is not an orthonormal basis,  $c$  by itself is. Thus, we can still easily transform back and forth between the whitened basis and the original data coordinates.

Note also that this implies that if we used PCA for dimensionality reduction, the data will also be whitened (if we standardize it afterwards). The first  $k$  principal components are always the same, no matter how many more we decide to compute afterwards. Thus, the PCA reduction just gives us the  $k$  most important dimensions of the PCA-whitened data.

## but wait...

Doesn't PCA have something to do with eigenvectors?  
Or singular value decompositions?

Only if you want to compute it *efficiently*.  
Or understand it even better. See [peterbloem.nl/blog/pca-2](http://peterbloem.nl/blog/pca-2)

If you've heard about PCA before, you may be wondering why I haven't discussed eigenvectors, or the singular value decomposition. These topics are only necessary if you want to **know the deeper workings of PCA**, and if you want to compute it efficiently.

For a basic understanding of what it does, all you need is a reconstruction loss, and gradient descent.

## PCA

### Principal Component Analysis

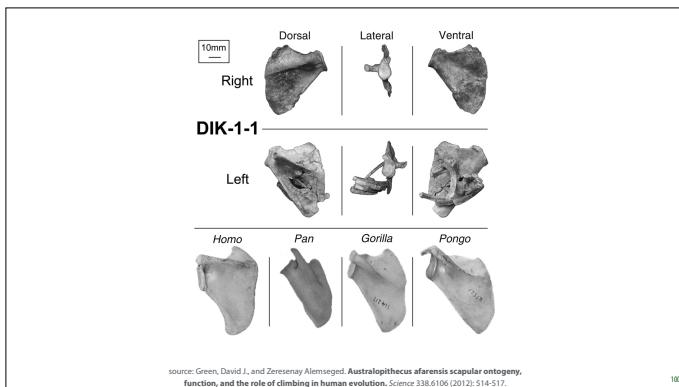
- linear transformation that minimizes reconstruction loss
- or: linear transformation that maximizes variance
- also: whitening transformation
- orders dimensions by variance captured/rec. loss

The first dimension is the most important for reconstructing the data, then the second and so on.

This may seem like a lot of math and complexity for something so simple as reducing the dimensionality of a dataset.

But it turns out that these principal components are actually extremely versatile, and can give us a lot of insight into our data.

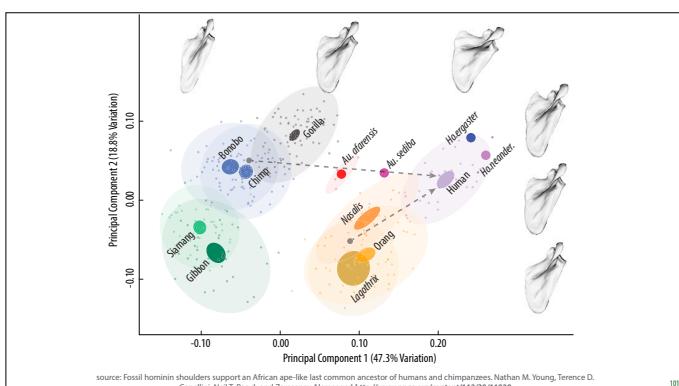
## So what's the point?



We'll start with an example of how PCA is often used in research. Imagine you're a palaeontologist, and you find a shoulder bone, belonging to some great ape.

If you are a trained anatomist specialising in primates, you can easily tell for a single shoulder bone whether it's an early hominin fossil, which is a very rare find, or a chimpanzee fossil which isn't rare. But how do you then substantiate this? "It's true because I can see that it is" is not very scientific.

image source: <https://science.sciencemag.org/content/338/6106/514.full>

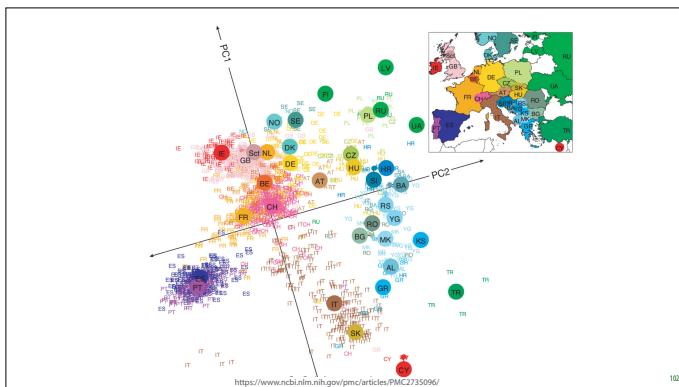


Here's one common approach. Take a large collection of the same specific bone (the *scapula*, or shoulder blade, in this case) from different apes and humans, and take a bunch of measurements (features) of each. Do a PCA, and plot the first two principal components. As you can see, the different species form very clear clusters, even in just two dimensions.

When we find a new fossil, we can see where it ends up in this space, and we can then show that what we've found is clearly closer to human than to chimp just by measuring it, and projecting it into this space.

Note also, that this data gives us some clues about how humans might have developed. The proto-humans *Australopithecus Afarensis* and *Australopithecus Sediba*, are both on a straight line between the cluster of Bonobos, Chimps and Gorillas on one side and modern humans on the other. These are indeed the great apes considered to be most like the ones from which we developed.

source: Fossil hominin shoulders support an African ape-like last common ancestor of humans and chimpanzees. Nathan M. Young, Terence D. Capellini, Neil T. Roach and Zeresenay Alemseged <http://www.pnas.org/content/112/38/11829>



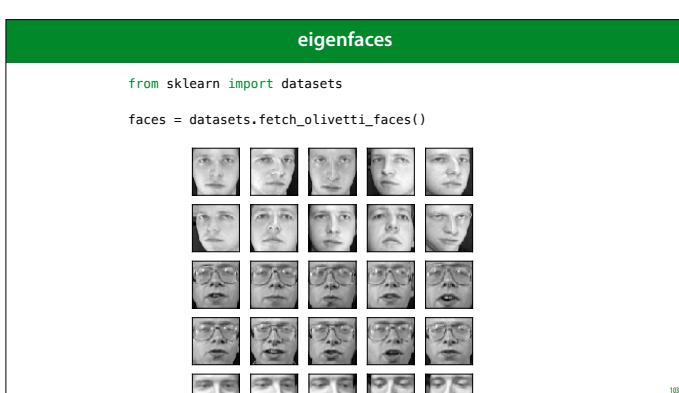
Here is another example of what PCA can tell us about a high-dimensional dataset.

In this research, the authors took a database of 1387 Europeans and extracted features from their DNA. They used about half a million sites on the DNA sequence where DNA varies among humans (i.e. 1387 instances: people, and 500k features: DNA markers). They also recorded where their subjects (or their immediate ancestors) were from.

Only the DNA data was fed to the PCA algorithm, with the person's origin only used afterward to color the points.

It turns out that the two principal components of this data largely express how far north the person lives, and how far east the person lives. This means that if you plot the data in the first two principal components, **you get a fuzzy picture of Europe**.

In short, the large scale geography of Europe can be extracted from our DNA. If I sent a large sample of European DNA to some aliens on the other side of the galaxy who'd never seen our planet, they could use it to get a rough idea of our geography.

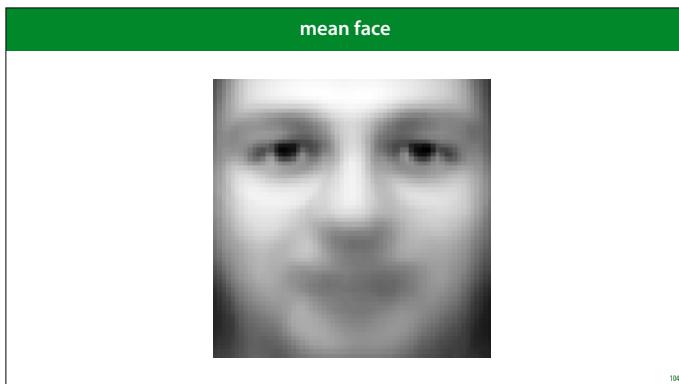


Finally, possibly the most magical illustration of PCA: **eigenfaces**.

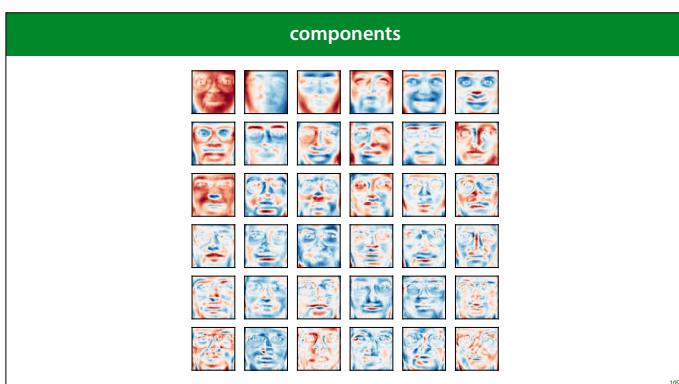
Here we have a dataset (which you can easily get from sklearn) containing 400 images, in 64x64 grayscale, of a number of people. The lighting is nicely uniform and the facial features are always in approximately the same place.

We take each pixel as a feature, giving us 400 instances each represented by a 4096-dimensional feature vector. Note that this essentially flattens the image into one long vector, ignoring the grid structure of the pixels.

*The prefix **eigen-** comes from the eigendecomposition often used to derive the PCA analysis. It's out of scope for us, but you should hopefully remember eigenvectors from your linear algebra. The eigenvectors of the covariance matrix **are the principal components**.*

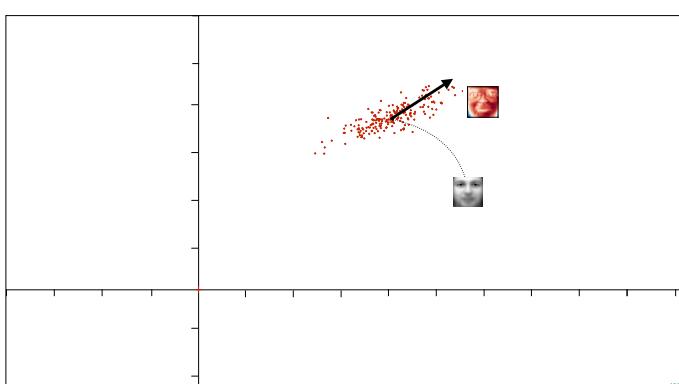


Here is the sample mean of our data, re-arranged back into an image.



Once we have the principal components, each a 4096-dimensional vector, we can take their values, assign them a color, like red for negative values, blue for positive values, and re-arrange them back into images. Remember, every dimension represents a pixel.

These are the first 30 principal components displayed this way (top left is the first, to the right of that is the second and so on).



Here is one way to interpret the principal components: the basis vectors that are most natural for our data. Remember, PCA is also a whitening operation.

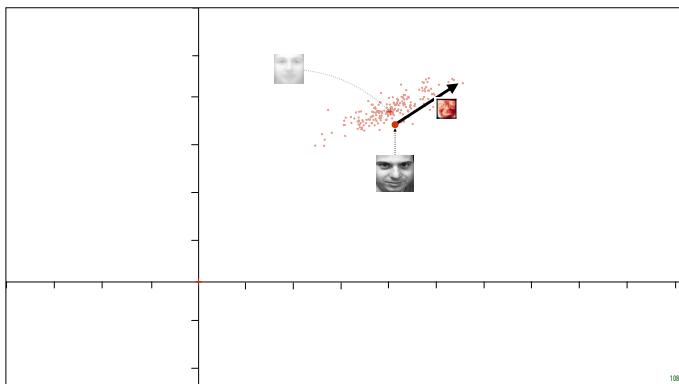
The first principal component is the direction that captures most of the variance of our data. Or, projecting our data down to the first principal component gives us the lowest reconstruction error.

We can visualize this space, by starting at the data mean, and adding a small bit of the nth principal component.



Starting from the mean face (in the middle column), we take little steps along the direction of one of our principal components (or in the opposite direction). These are the first five.

We see that moving along the first principal component roughly corresponds to ageing the face. Moving along the fourth seems to make the face more feminine.

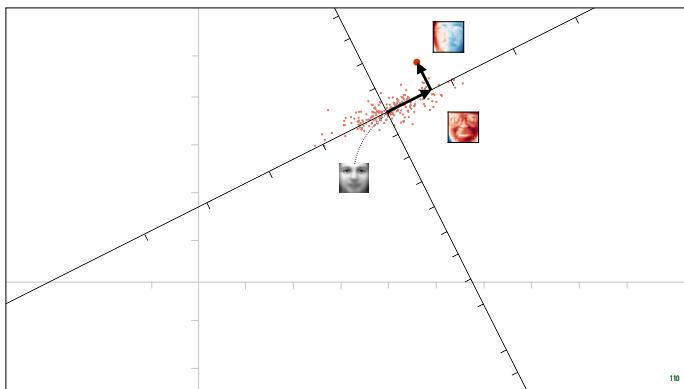


Instead of starting at the mean face, we can also start at some other point, like one of our instances, and add or subtract small bits of the principal components.

*The reason that we can add the principal components directly to the data like this is that the reduction and reconstruction are linear operations. If we use nonlinear versions of PCA, this trick won't work anymore. Details in the reading materials.*



The middle column represents the starting point. To the right we add the  $k$ -th principal component, to the left we subtract it. Note, in particular the effect of the fifth principal component: subtracting it opens the mouth, and adding it seems to push the lips closer together.

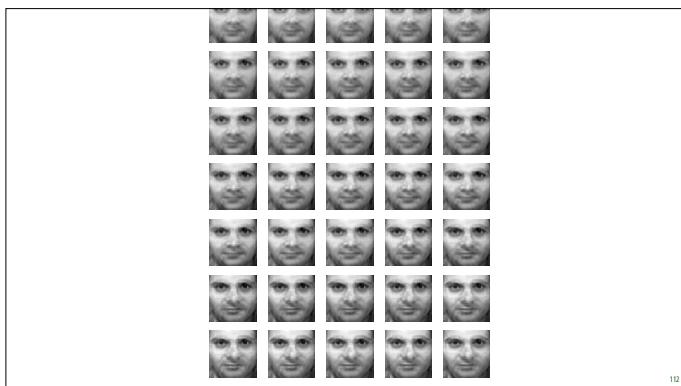


To reconstruct a point, we start with the mean, and add a bit of the first principal component, then of the second principal component and so on.

If we think of our principal components as a new **basis** for our data, then we are just looking up our point by first moving some distance along the first axis, then along the second axis and so on. Just like we would look up a point given its coordinates in the standard basis.



Here's what that looks like. Top left is the mean. To the right of it is the reconstruction from just the first principal component. Next is what we get if we add the second principal component to that and so on.



After 60 principal components out of a possible 4096, the image starts to look pretty recognizable. We've reduced our data from 4096 dimensions to just 60 dimensions, and still retained enough information to tell people apart.

## recap

**PCA:** dimensionality reduction

### Using linear transformations

We'll discuss nonlinear dimensionality reductions in lecture 9

### Principal components are a *natural basis* for your data

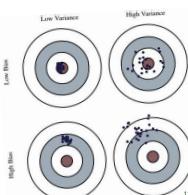
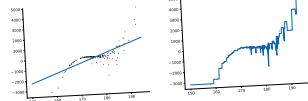
Good way of getting insight into your data

13

## bias and variance

**Feature expansion:** more expressive model, lower bias, higher variance, **more overfitting.**

**Dimensionality reduction:** reduce model expressivity, higher bias, lower variance, **less overfitting.**

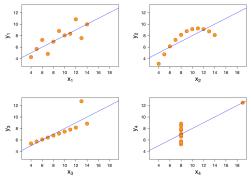


14

One final thought to consider: the bias/variance view can cast a new light on the topics we've seen today. Imagine if you have a dataset and you train a linear model on it. We've seen that a linear model can be perfectly optimized by analytical means, so this should be a very efficient approach.

We've also seen that you can expand its features to make the model more expressive. For instance, this allows a linear model to take the shape of a parabola in the original feature space. We are increasing the model expressivity by doing this. This means that the bias will shrink, because the model will fit the data better, but also that we risk an increase in variance, because the model may begin to overfit.

The key insight here is that dimensionality reduction, like that provided by PCA, is the opposite of this. If we start with a lot of features already (like all the pixels in an image), then even a simple linear model may already overfit, and put us in the high variance regime. PCA allows us to reduce the features to a mixture that retains only the crucial information: we take information from all features, but we throw away the noise that the model would otherwise overfit on. This pushes us away from the high variance regime, and, if we go too far, towards the high bias regime.



mlcourse@peterbloem.nl